Real-time Neuromorphic Visual Pre-Processing and Dynamic Saliency

by

Jamal Lottier Molin

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

March, 2017

© Jamal Lottier Molin 2017

All rights reserved

Abstract

The human brain is by far the most computationally complex, efficient, and reliable computing system operating under such low-power, small-size, and light-weight specifications. Within the field of neuromorphic engineering, we seek to design systems with facsimiles to that of the human brain with means to reach its desirable properties. In this doctoral work, the focus is within the realm of vision, specifically visual saliency and related visual tasks with bio-inspired, real-time processing. The human visual system, from the retina through the visual cortical hierarchy, is responsible for extracting visual information and processing this information, forming our visual perception. This visual information is transmitted through these various layers of the visual system via spikes (or action potentials), representing information in the temporal domain. The objective is to exploit this neurological communication protocol and functionality within the systems we design. This approach is essential for the advancement of autonomous, mobile agents (i.e. drones/MAVs, cars) which must perform visual tasks under size and power constraints in which traditional CPU or GPU implementations to not suffice. Although the high-level objective is to design a

complete visual processor with direct physical and functional correlates to the human visual system, we focus on three specific tasks.

The first focus of this thesis is the integration of motion into a biologically-plausible proto-object-based visual saliency model. Laurent Itti, one of the pioneers in the field, defines visual saliency as "the distinct subjective perceptual quality which makes some items in the world stand out from their neighbors and immediately grab our attention." From humans to insects, visual saliency is important for the extraction of only interesting regions of visual stimuli for further processing. Prior to this doctoral work, Russel et al. [1] designed a model of proto-object-based visual saliency with biological correlates. This model was designed for computing saliency only on static images. However, motion is a naturally occurring phenomena that plays an essential role in both human and animal visual processing. Henceforth, the most ideal model of visual saliency should consider motion that may be exhibited within the visual scene. In this work a novel dynamic proto-object-based visual saliency is described which extends the Russel et. al. saliency model to consider not only static, but also temporal information. This model was validated by using metrics for determining how accurate the model is in predicting human eye fixations and saccades on a public dataset of videos with attached eye tracking data. This model outperformed other state-of-the-art visual saliency models in computing dynamic visual saliency. Such a model that can accurately predict where humans look, can serve as a front-end component to other visual processors performing tasks such as object detection and

recognition, or object tracking. In doing so it can reduce throughput and increase processing speed for such tasks. Furthermore, it has more obvious applications in artificial intelligence in mimicking the functionality of the human visual system.

The second focus of this thesis is the implementation of this visual saliency model on an FPGA (Field Programmable Gate Array) for real-time processing. Initially, this model was designed within MATLAB, a software-based approach running on a CPU, which limits the processing speed and consumes unnecessary amounts of power due to overhead. This is detrimental for integration with an autonomous, mobile system which must operate in real-time. This novel FPGA implementation allows for a low-power, high-speed approach to computing visual saliency. There are a few existing FPGA-based implementations of visual saliency, and of those, none are based on the notion of proto-objects. This work presents the first, to our knowledge, FPGA implementation of an object-based visual saliency model. Such an FPGA implementation allows for the low-power, light-weight, and small-size specifications that we seek within the field of neuromorphic engineering. For validating the FPGA model, the same metrics are used for determining the extent to which it predicts human eye saccades and fixations. We compare this hardware implementation to the software model for validation.

The third focus of this thesis is the design of a generic neuromorphic platform both on FPGA and VLSI (Very-Large-Scale-Integration) technology for performing visual tasks, including those necessary in the computation of the visual saliency. Visual pro-

cessing tasks such as image filtering and image dewarping are demonstrated via this novel neuromorphic technology consisting of an array of hardware-based generalized integrate-and-fire neurons. It allows the visual saliency model's computation to be offloaded onto this hardware-based architecture. We first demonstrate an emulation of this neuromorphic system on FPGA demonstrating its capability of dewarping and filtering tasks as well as integration with a neuromorphic camera called the ATIS (Asynchronous Time-based Image Sensor). We then demonstrate the neuromorphic platform implemented in CMOS technology, specifically designed for low-mismatch, high-density, and low-power. Such a VLSI technology-based platform further bridges the gap between engineering and biology and moves us closer towards developing a complete neuromorphic visual processor.

Primary Reader: Professor Ralph Etienne-Cummings

Secondary Reader: Professor Andreas Andreou

Acknowledgments

There are many people and organizations that I would like to acknowledge who provided support in many ways while pursuing this research and completing these various projects.

First of all, I would like to acknowledge my advisor, Dr. Ralph Etienne-Cummings for all of his support. He welcomed me to his lab, first when I was interning in the summer of 2008. He allowed me to get quality research experience during my internship experience. He later accepted me into his lab as a Ph.D. student in 2011. He was always very supportive in regards to funding as well as my research work. He gave me space to think on my own and develop new ideas but at the same time was always available for questions and advice. He helped me to think about problems with different perspectives in ways I had not thought about before. I am grateful that he was my PhD advisor, he is someone I look up to, and I hope to continue to collaborate with him and the lab in the future.

I also would like to acknowledge Dr. Andreas Andreou who was like a co-advisor for me. I worked closely with him and his lab under the DARPA UPSIDE project

from which much of the IFAT-related work stemmed from. He always had honest and great advice regarding my work. I'm thankful for being able to work with him and his lab while completing this thesis work.

I would also like to thank Dr. Ernst Niebur for his assistance and support of my work regarding visual saliency. He was a crucial part of this work in giving the neuroscience perspective of the work and in coming up with new ideas.

It is also important to acknowledge Dr. Philippe Pouliquen for all of the knowledge he has provided regarding many things whether it be related to software installation, circuit design, or even car issues. He always made himself available for questions and I have learned many things from him.

Dr. Garrick Orchard (who is currently at SINAPSE in Singapore) was also of great support as he was my mentor when I interned in 2008 and was also finishing up his Ph.D. when I first started in 2011. He was a close mentor and always gave great advice related to my work and just anything related to the path in pursuing Ph.D. He does related work so I received a lot of great advice and knowledge from him regarding FPGAs, ATIS camera, Cadence, and many other topics while he was here in the lab and even after going to Singapore.

I'd also like to thank many other colleagues and collaborators including the members in Dr. Andreas Andreou's lab. I'd especially like to thank Tomas, Kayode, Gaspar, Martin and Philippe for the all-nighter chip submits and making them enjoyable. I'd like to thank Ernst Niebur's lab for all of their support and feedback

with the visual saliency work and related MURI work. On that note, it is important to acknowledge all of the ONR MURI professors and members for all of their feedback and discussion with the dynamic visual saliency model. It is also important to acknowledge organizations including Draper Lab, BAE Systems, and Telluride Neuromorphic Workshop for all of the people I have met and the knowledge I have gained from these organizations.

I would like to acknowledge all of my labmates both past and present for making the lab environment a fun place to work and also for always being available for discussion about various research topics. They have also been great friends even outside the lab. These include Jack Zhang, Adam Khalifa, Tao Xiong, Kerron Duncan, John Rattray, Prof. Chetan Thakur, Dr. Webert Montlouis, Bayo Eisape, Hyungmu Lee, Qi Wang, Sherry Chiu, Duncan Parke, Dr. Christian Brandli, Meg Chow, Prof. Mariel Alfaro, Martina Leistner, Eleonora Costa, Vigil Varghese, Dr. Kevin Mazurek, Dr. Alex Russell, Dr. Garrick Orchard, Dr. Andre Harrison, Dr. Xiaoyu Guo, Dr. Fope Folowosele, Dr. Guy Hotson, Dr. Robert Yaffe.

Lastly, but definitely not least, I would like to acknowledge my family for all of their support throughout the highs and lows of my Ph.D. experience. I could not have accomplished any of this without them. I especially thank my mom (Yvette), my dad (Al), my sister (Jasmine), and finally my wife (Britney) and family. They have always encouraged me and supported me throughout this process.

I thank God for who He is and giving me strength through the ups and downs

and I thank Him for putting my wife into my life and for her continuous support.

Dedication

This thesis is dedicated to my wife (Britney), my mom (Yvette), my dad (Al), my sister (Jasmine), my grandparents (Gigi, Poppy, Mama Yotte, and Papa Leonce), and aunts, uncles, and cousins.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In the science-fiction film, The Terminator (1984), actor Arnold Schwarzenegger plays a cyborg (the "Terminator") capable of effortlessly interacting with its environment and humans in order to fulfill its purpose to prevent machines from being destroyed in a post-apocalyptic future. The Terminator is capable of receiving information from the world, processing the information, making decisions based on the information, and reacting based on those decisions. In a sequel to this box-office hit, the Terminator reveals what it is that makes it "tick" by stating: "My CPU is a neural net processor... a learning computer". Although merely a science-fiction film, imagine if we could design such a processor that can "think" and "reason"; a processor with structure and functionality that have facsimiles to the human brain.

Figure 1.1: The Terminator film featuring Arnold Schwarzenegger plays a cyborg capable of interacting with its environment and other humans. When asked what makes him "tick", he states "My CPU is a neural net processor... a learning computer". Like "The Terminator", in this research, the objective is to design hardware-based processors inspired by the human brain capable of performing visual tasks.

Having an average weight of 1.3 kg and volume of 1130 cm^2 in men and 1260 cm^2 in women [31,32], the human brain is capable of effortlessly and rapidly processing information obtained by our sensory inputs (smell, auditory, vision, taste, and touch). In doing so it consumes merely 20 watts of power. Although all senses are important and necessary for designing a complete neuromorphic system which emulates the human brain and can perform cognitive tasks, we focus on vision in this research. Pre-attentively, humans can process an overwhelming amount of dynamic visual information from the retina via the optic nerves in real-time. We can perform cognitive tasks including detecting, recognizing, and even tracking objects once attended to. Furthermore, these visual tasks are performed while natural phenomena such as ego-motion and noise/distractions are present. While biological systems can instinctively perform these complex tasks under low-power, small-size, and light-weight constraints, engineered systems struggle to perform such tasks under these desirable specifications.

Continuing advancements in microchip technology has followed Moore's Law [33], which predicted that the number of transistors that can be manufactured on a single microchip doubles every two years. This allows for smaller, faster, and more energy-efficient transistors. However, the cost of building facilities capable of manufacturing such technology is rising exponentially. Therefore, it is not only important that the transistor size is decreasing, but more advancement in how we use the technology in designing computing systems is equally significant. The human brain consists of

$\sim$100 billion neurons and $\sim$100 trillion synapses. It is capable of performing visual (and other sensory) tasks through the synaptic connectivity and communication between neurons via spikes (or action potentials). The underlying stochasticity of spike streams in computing and representing information makes the system robust to noise. Henceforth, the computing systems we design should also operate in a similar spike-based manner and contain structure with physical properties similar to neurons and their connectivity. A merging of the advancement in technology and advancement in design methodology is how the low-power, light-weight, and small-size specifications can be met.

These neuromorphic systems have a wide-range of applications. There has been recent research in designing visual processors which can be coupled with autonomous mobile systems such as drones, micro aerial vehicles (MAVs), and self-driving cars for efficient navigation, object detection, and object recognition [34–37]. These autonomous robotics are necessary for search-and-rescue, disaster response, and surveillance missions [38]. However, many of the current approaches utilize traditional von Neumann approaches via CPUs or GPUs for processing. This is not ideal as such processors are typically larger in size, slower in speed, and power-hungry.

Another interesting application of this work is within the field of virtual reality (VR). VR systems are being explored in neuroscience research to simulate natural events and social interactions [39]. It is also being used in the field of robotics for analyzing interactions between the visual systems of a robotic agent and the environ-

ment for properly solving the visual tasks necessary to interact with the 3-dimensional world [40]. As research in VR continues to quickly progress, such neuromorphic visual processors can be integrated with VR systems for applications such as these, as well as providing augmented vision within a virtual, multi-dimensional environment.

Finally, by designing systems that mimic the human brain, they can serve as a platform on which neuroscientists can use to develop hypotheses, advancing research in neuroscience as well.

These are all possible applications which support the necessity of designing bio-inspired systems performing visual tasks in real-time under low-power, small-size, and light-weight constraints. The design of bio-inspired, neuromorphic sytems is a broad topic, so, the specific aims of this doctoral work will be discussed in the proceeding section (Section 1.2).

1.2 Overview/Aim of Thesis

The broad scope of this research is to design systems in hardware capable of performing visual tasks in real-time with physical characteristics and functionality that is similar to the human visual system ("the gold-standard").

However, building a complete system that operates identically to the visual system of the human brain is beyond the scope of this doctoral work. Instead, we focus on two specific objectives which contribute to this long-term goal. The first is a

real-time computational model of dynamic visual saliency. The second is a generic system capable of performing image processing tasks which can also be utilized by the visual saliency model, specifically, image filtering and dewarping. The diagram in Fig. 1.2 gives a visual understanding of the objectives of this doctoral work and how it falls under the umbrella of the broad scope of this work. We take advantage of VLSI CMOS technology and FPGAs in designing a real-time dynamic visual saliency model and a generic bio-inspired visual processor capable of performing filtering and dewarping tasks. The most ideal visual processor, the human visual system, serves as our reference in designing these neurompрhic systems.

In the proceeding sections, a foundation to this research is established by providing an overview to the human visual system in Section 1.3. A background to neuromorphic engineering will be given in Section 1.4. Dynamic visual saliency plays a key role in this doctoral work, and therefore, a brief background to visual saliency from a biological perspective and an engineering perspective will be given in Section 1.3.2 and Section 1.5.3, respectively.

1.3 Human Visual System

1.3.1 Eye Anatomy and Visual Processing

The book "Neuroscience" by Purves et al. states:

"The human visual system is extraordinary in the quantity and quality of

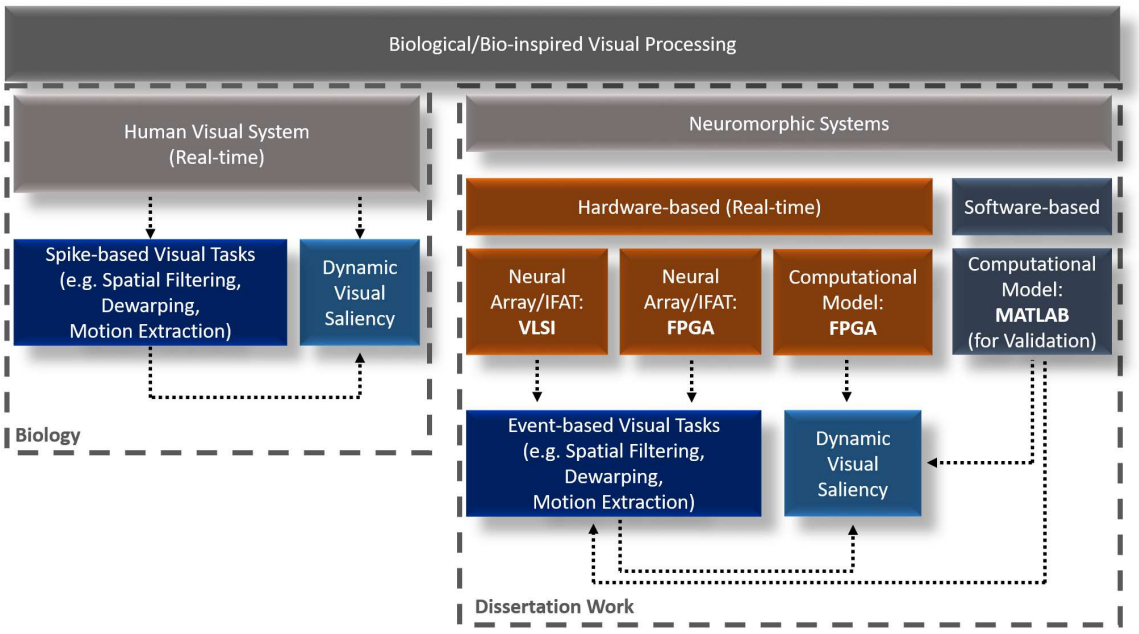


Figure 1.2: Diagram showing organization and field of research of this doctoral work.

> *information it supplies about the world. A glance is sufficient to describe the location, size, shape, color, and texture of objects and, if the objects are moving, their direction and speed. Equally remarkable is the fact that viewers can discern visual information over a wide range of stimulus intensities, from faint light of stars at night to bright sunlight."* [4]

This statement accurately illustrates the capabilities of the human visual system. It effortlessly extracts visual information from the world and performs complex tasks within milliseconds under low-power and small-size constraints.

This visual information is extracted from the world via the eyes. A simplified diagram of the anatomy of the human eye can be seen in Fig. 1.3. The eye is a fluid-filled sphere consisting of three layers of tissue. The inner most layer is the retina, consisting of light-sensitive neurons responsible for capturing and transmitting visual signals. The inner surface of the retina is called the fundus. It consists of the optic

disk, a whitish, circular area where retinal axons leave the eye and travel through the optic nerve to cortex. The optic disk contains no photoreceptors (producing the phenomena known as the "blind spot"). The fundus also contains the macula lutea, an oval spot containing yellow pigament located near the center of the retina. It supports high visual acuity, resolving fine detail. The center of the macula is the fovea and contains the greatest acuity.

The adjacent layer, the uveal tract, is responsible for nourishing the photoreceptors of the retina. It contains the iris, (the colored part of the eye) responsible for allowing the pupil (opening at center of the eye) to be adjusted under neural control. This is similar to the functionality of the diaphragm of a camera, controlling the amount of light that passes through by controlling the aperture. The outermost layer is the sclera, a white, tough tissue consisting of the cornea, the transparent tissue permitting light to enter the eye. This is analogous to the lens of a camera.

The cornea and lens of the eye are transparent, optical components which contribute towards allowing light to pass through to the retina. They are responsible for the refraction of light necessary for the formation of focused images on the photoreceptors of the retina. These dynamic changes in the refractive power of the lens is called accommodation. It is the automatic adjustment of the lens for viewing near and distant objects. Adjustment to the size of the pupil contributes to the clarity of the image.

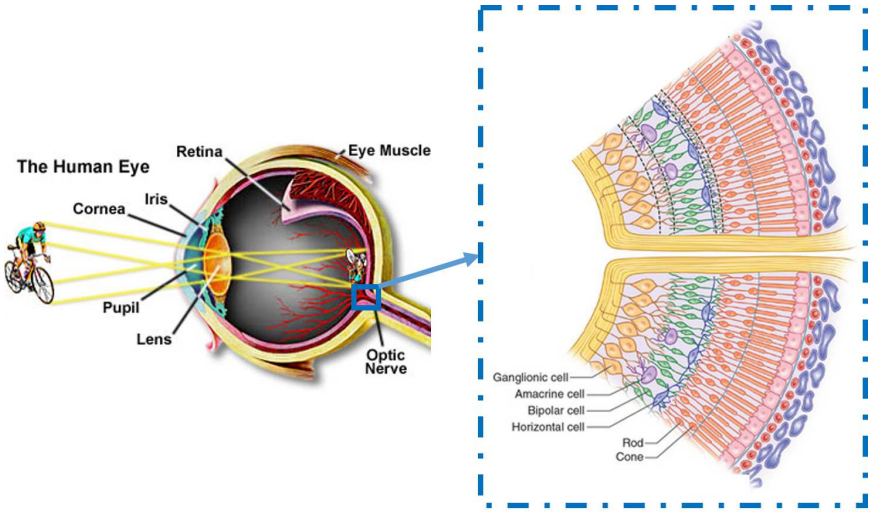


Figure 1.3: Brief anatomy of the human eye (left) and retinal layers (right). The rods and cones are photoreceptors which are activated by light. These photoreceptors have synaptic connections to interneurons (horizontal, bipolar, and amacrine cells). Finally, these interneurons connect to ganglion cells with center-surround receptive fields. Axons of ganglion cells form the optic nerve bundle which transmits signals to further layers of visual processing. See Section 1.3 for details. Image from [2, 3].

The retina contains photoreceptors (photosensitive neurons) which converts electrical energy generated from light into action potentials which travel via axons to the optic nerve. There are five types of neurons in the retina: photoreceptors, bipoloar cells, ganglion cells, horizontal cells, and amacrine cells. These cells are arranged in layers (See Fig. 1.3). Light first hits the photoreceptors. There are two types of photoreceptors, rods and cones. Rods are responsible for vision in the dark as they respond well to dim light. Rods are found mostly in peripheral regions of the retina. Cones are concentrated in the central region of the retina (fovea). They are responsible for high acuity tasks and enable us to perceive color. These rods and cones have synaptic terminals that connect to bipolar and horizontal cells. The shortest electrical path to the optical nerve is from photoreceptors to bipolar cells to ganglion cells. The axons of ganglion cells form the optic nerve bundle. Horizontal cells enable lateral connections between photoreceptors and bipolar cells. They help to maintain sensitivity to contract over various luminance. The amacrine cells have presynaptic connections from bipolar cells and postsynaptic connections to dendrites of ganglion cells. Each retinal neuron has a receptive field formed from the complex connectivity between these neurons. The receptive field determines what activates the neuron. Stephen Kuffler discovered that ganglion cells (from the cat retina) have a center-surround receptive field [41]. There exists ganglion cells with both on-center and off-center receptive fields, and approximately equal of each. Ganglion cells with on-center receptive fields have an excitatory center region with an inhibitory circular

surround region. They are activated when there is a light spot at the center of its receptive field or when there is an increase in luminance. Ganglion cells with off-center receptive field have an inhibitory center region with an excitatory circular surround region. They are activated when there is a dark spot at the center of its receptive field or when there is a decrease in luminance. The center-surround receptive field of ganglion cells and their activation can be seen in Fig. 1.4.

Ganglion cell axons bundle together to form the optic nerve exiting the retina. These axons target various structures but primarily projects to the lateral geniculate nucleus (LGN) in the thalamus in the midbrain area. From the LGN, signals are then transmitted to the striate cortex, better known as the primary visual cortex (V1). See Fig. 1.5. There are two parallel streams of processing from the LGN to the V1 in the back of the brain; the magnocelluar and parvocelluar pathways. The magnocelluar pathway contains information related to contrast and motion and consists of larger neurons (larger receptive fields). The parvocelluar pathway contains information related to fine structure and color and consists of smaller neurons (smaller receptive fields). These pathways are also known as the dorsal (magnocelluar) and ventral (parvocelluar) streams.

The dorsal stream (the "where" or "how" pathway) contains motion information and allows us to determine where objects are in space, and guides the movement of our eyes and arms. Neurons from V1 project to V2 and then project to V5/MT (Middle Temporal). V1 neurons also project directly to MT. MT layer consists of

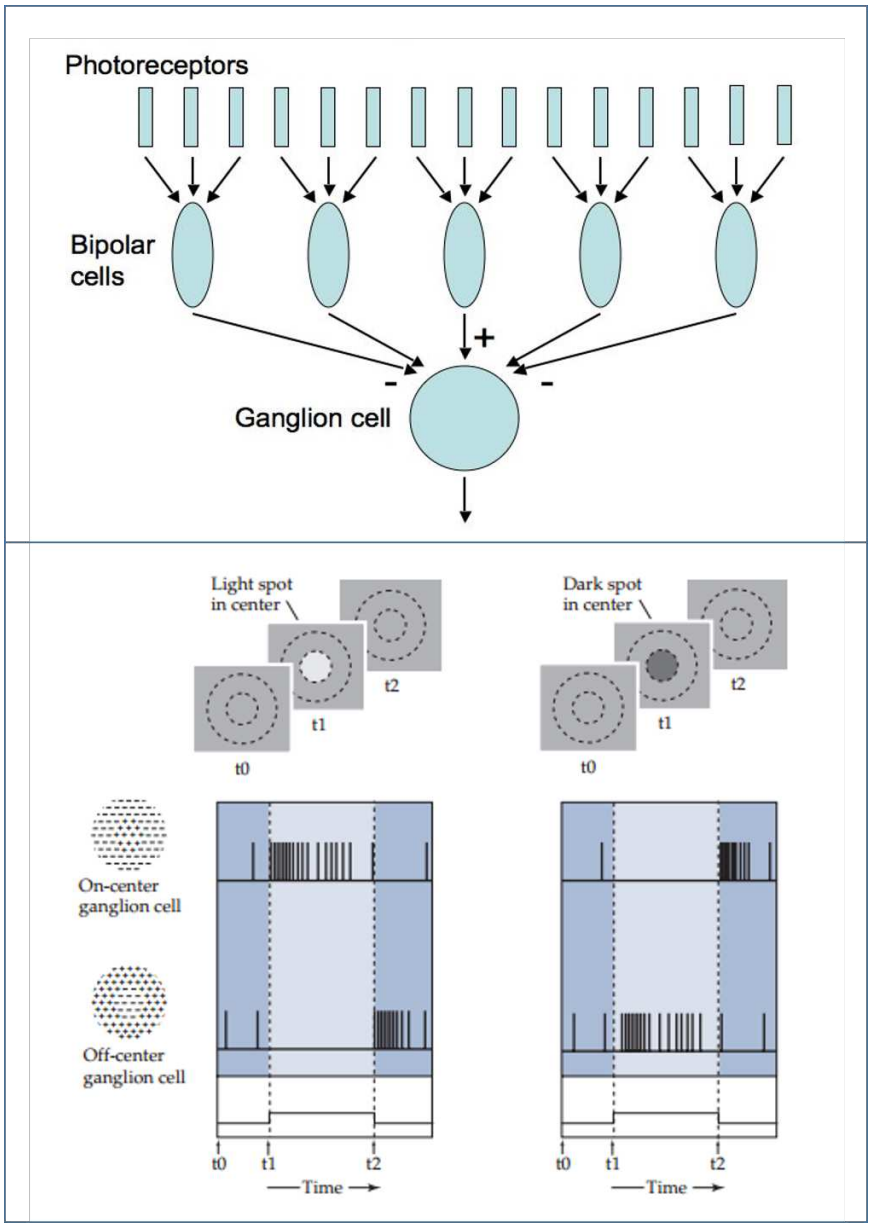


Figure 1.4: The center-surround receptive field of ganglion cells. Visual representation of on-center receptive field on the left. On-center ganglion cells respond to a light spot at the center of its receptive field or an increase in luminance. The off-center ganglion cells respond to a dark spot at the center of its receptive field or a decrease in luminance. Images from [4, 5].

neurons selective to specific directions of stimulus motion. Neurons also project to MST (Middle Superior Temporal) area. It has been shown that first order motion is processed in V1 and V2 while second order motion is processed in MT and MST [42].

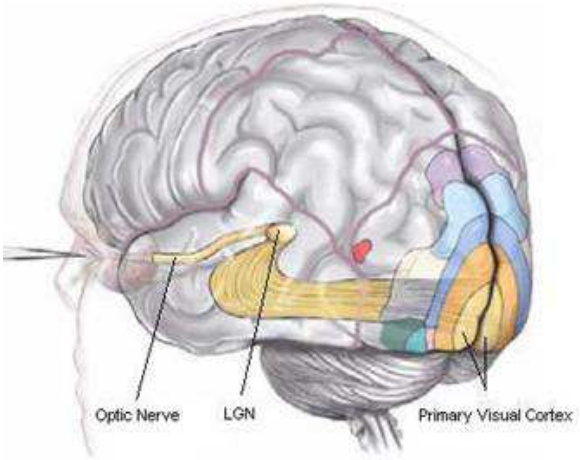


Figure 1.5: Ganglion cell axons from the retina project to the lateral geniculate nucleus (LGN) of the thalamus. From the LGN, signals project to the primary visual cortex (V1) along two parallel pathways (magnocelluar and parvocellular pathways). Image from [6].

The ventral stream (the "what" pathway) contains information related to shape, size, and recognition of objects we see. It consists of neurons in V1 and V2 projecting to V4 and IT (inferior temporal cortex). Neurons in V2 and V4 are tuned to orientation, spatial frequency, and color.

It is also important to note the retinotopy of neurons in the primary visual cortex. There exists a point-to-point mapping between the retina and V1. It allows positioning of the visual world onto a 2-D retinotopic map projected to the primary visual cortex. Depth information evolves from information from the two eyes (similarly to

triangulation). Neurons in V1 are organized in columns in which each column of neurons is activated by a particular oriented edge. These neurons are called simple cells and have a Gabor-filter type receptive field consisting of excitatory and inhibitory regions. Complex cells are found in V1/V2/V3 and also respond to oriented edges as well (and can have synaptic connections to simple cells), but have a more non-linear receptive field by being invariant to location and scale.

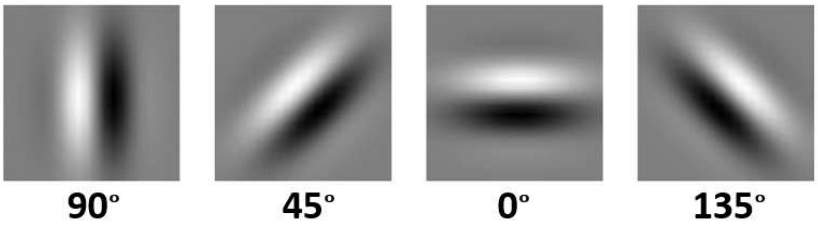


Figure 1.6: Gabor filter type receptive field of simple cells in V1. Dark areas represent inhibitory regions and light areas represent excitatory regions. These receptive fields are used for extracting edge information. Image from [7].

This introduction to the human visual system provides a foundation for which this work is built on. The work in this thesis is inspired by the physical characteristics and functionality of the human visual system.

1.3.2 Visual Saliency: A Biological Perspective

In the previous section, an introduction the human visual system as a whole was introduced. In this section we focus on how visual saliency is computed within the

human visual system. The computation of dynamic visual saliency is a vital component of this work, thus, an introduction to visual saliency and how it is computed from a biological perspective is necessary.

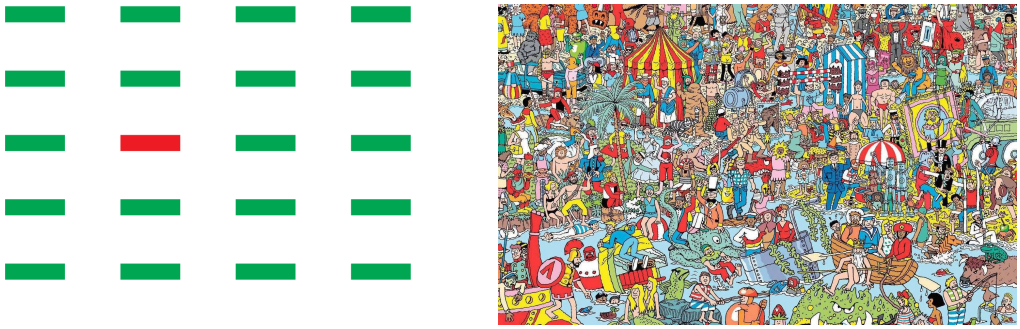
Laurent Itti, a pioneer within the field, accurately defines visual saliency as:

> *"The distinct subjective perceptual quality which makes some items in the world stand out from their neighbors and immediately grab our attention."* [43]

In humans, each optic nerve contains an average of 10^6 retinal ganglion axons each transmitting information to the brain [44]. Considering the transmission rate of these retinal ganglion cells, this equates to the brain receiving on the order of 100 Mbps of spatial and temporal visual information per optic nerve [45]. Processing this overwhelming amount of information in parallel and in real-time is simply impossible for any human brain due to the difficulty of such a task [46]. To overcome this complexity, we instead utilize selective attention and attend only to regions of the visual stimuli deemed interesting, or salient. It is these salient regions that are attended to and forwarded to the proceeding stages of visual processing.

There are two components which contribute to visual saliency, bottom-up and top-down. Bottom-up saliency is a function of only the inherent properties of the visual stimulus itself. If you look at Fig. 1.7A, the red rectangle immediately grabs your attention merely because of its unique color compared to the color of the surrounding rectangles. Bottom-up saliency is computed rapidly in early stages of the human visual system. On the contrary, top-down saliency is a function the viewers biases

based on internal state and goals. Top-down saliency strongly modulates bottom-up saliency. If you are looking at a "Where's Waldo" image (See Fig. 1.7B), nothing may be inherently salient until you are told to look for Waldo. Now, any object in the image which may resemble Waldo's shape or color (red and white) will grab your attention. In this doctoral work, we focus primarily on bottom-up visual saliency.



(A) (B)

Figure 1.7: Images A and B demonstrate the difference between bottom-up and top-down visual saliency, respectively. In Image A, the red rectangle immediately grabs your attention because the features of the stimuli within the image. The red color is unique to its surrounding (green rectangles), and therefore, attention is focused on the red rectangle. In Image B, we give the task to find "Waldo", a man with glasses and a red and white-striped shirt. Once given this prior information, top-down saliency is applied and we attend to regions of the image with properties similar to Waldo (e.g. the red and white-striped umbrella in the right half the image.

As previously stated, bottom-up saliency is computed in the early stages of visual processing. There are two related theories to how visual saliency is computed in these

early stages of the visual system, feature-based and object-based visual saliency.

1.3.2.1 Feature-based Visual Saliency

The first, and original theory is feature-based visual saliency. Early psychophysical and physiological studies suggests that there exists evidence of selective attention in the human visual system which is feature-driven. Pyschophysical experiments by Triesman and Gelade (1980) explained a two-stage hypothesis of feature-based attention [47]. In the first-stage, visual search for targets defined by a single feature is processed preattentively, rapidly, and in parallel across the visual field ("popout effect"). In the second stage, search for a conjunctive target defined by many features requires attention and is a serial process. Elementary features are processed in parallel, including color, orientation of edges, intensity, and first-order motion. Physiologically, Goldberg and Wurtz found neurons in the superficial layers of the superior colliculus of monkeys that respond to visual stimuli if the monkey intended on using its receptive field as the location of its next saccade [48]. Other structures such as the posterior parietal cortex [49], frontal eye fields [50], and visual cortex in V1 [51] have been suggested to contribute to computation of visual saliency.

Over the past few decades, Koch and Ullman [52], Milanese et al. [53], Wolfe [54], Niebur and Koch [55], and others have conducted research, utilizing psychophysical and physiological evidence, for understanding how feature-based, bottom-up saliency is computed in the brain. Neurons in early stages of visual processing are sensitive

to elementary features including color, orientation, intensity, and motion. Each of these features are computed independently, and in parallel across the visual field. Furthermore, these features are computed at various spatial and temporal resolutions. This gives rise to multiple parallel topographic maps known as "feature maps" (See Fig. 1.8). Spatial locations are preserved from the visual stimulus throughout the extraction of features. Independently within each feature map, there exists lateral inhibition such that neural activity is increased at locations which differ from their surrounding (center-surround receptive field of cells in early stages of visual processing). This results in parallel topographic "conspicuity" maps for each feature. For example, with respect to color, from Fig. 1.7A, the location of the red rectangle among the green rectangles will be more conspicuous than any of the green rectangles surrounded by other green rectangles. Furthermore, a normalization operation which uses long-range inhibition within each feature, independently. Neural responses to visual information within its receptive field is modulated by surrounding neural responses to the same feature within its receptive field. The neural response is increased when it differs from the surrounding neural responses and supressed when it is similar. This is supported by the non-classical receptive field inhibition idea presented by Allman et al. [56]. These conspicuity maps are then combined forming a final non-topographic saliency map depicting the amount of conspicuity at each location independent of feature or spatial/temporal scale. This saliency map is a 2-dimensional retinotopic map in which locations of high activity represent high saliency

and locations of low activity represent low saliency (See Fig. 1.8).

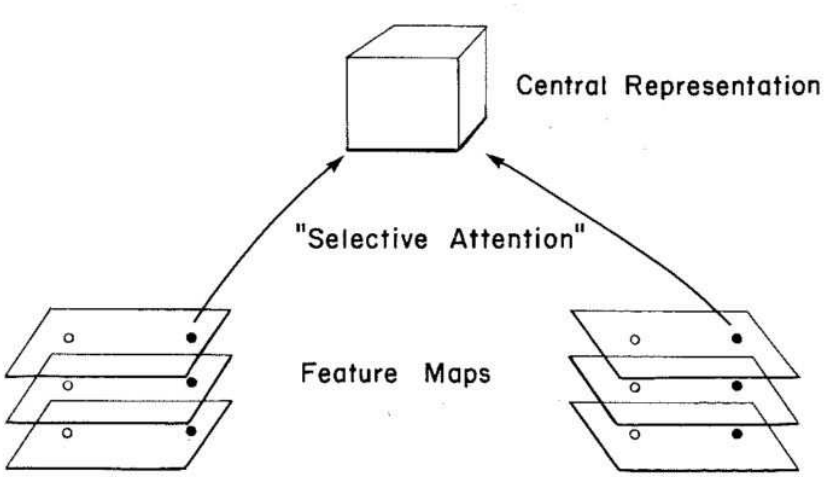


Figure 1.8: General visual representation of saliency computation. The input visual stimulus gives rise to multiple parallel feature maps. Within each feature map, conspicuity is computed, enhancing locations which are unique with respect to its surrounding location. These conspicuity maps are combined to form the final central representation, the saliency map. This forms our "selective attention".

1.3.2.2 Object-based Visual Saliency

Contrary to feature-based saliency, more recent studies suggest that objects are in fact perceived prior to features. This supports Gestalt psychology which states that the whole of an object is perceived before its individual features are processed [57]. The perception of object is based on grouping features based on Gestalt factors of proximity, similarity, good continuation, closure, and symmetry of order. This idea is supported by the work of Neisser (1967) which proposed that perceptual analysis

consists of two stages [58]. The first is the preattentive stage in which the visual field is organized into separate objects based on these Gestalt principles. The second stage requires focused attention to analyze a particular object in more detail. The preattentive stage is a parallel process that happens rapidly across the visual field. The second stage is a serial process, requiring attention, and is responsible for our limited ability to see more than only a few objects at once.

Building on Neisser's work, one theory of how object-based saliency is computed is the integrated competition hypothesis by Duncan [59]. Similarly to Neisser, is supports the idea that preattentively, objects are computed followed by focused attention which allows us to see the details of an object. Duncan takes this one step further and shows that two judgments concerning the same object can be made simultaneously while two judgments concerning different objects cannot be made simultaneously. Therefore, objects computed preattentively compete for limited resources across all sensorimotor systems. When one object is attended to, it gains dominance for utilization of these resources while utilization by other objects is inhibited.

Another theory of how object-based visual saliency is computed is based on the coherence theory by Rensink [8]. It is based on the notion of proto-objects as seen in Fig. 1.9. The coherence theory states that there exists low-level proto-objects which are formed rapidly and in parallel across the visual field. These proto-objects are preattentive structures with limited spatial and temporal coherence. Focused attention is required to stabilize a small number of proto-objects, henceforth, generating the

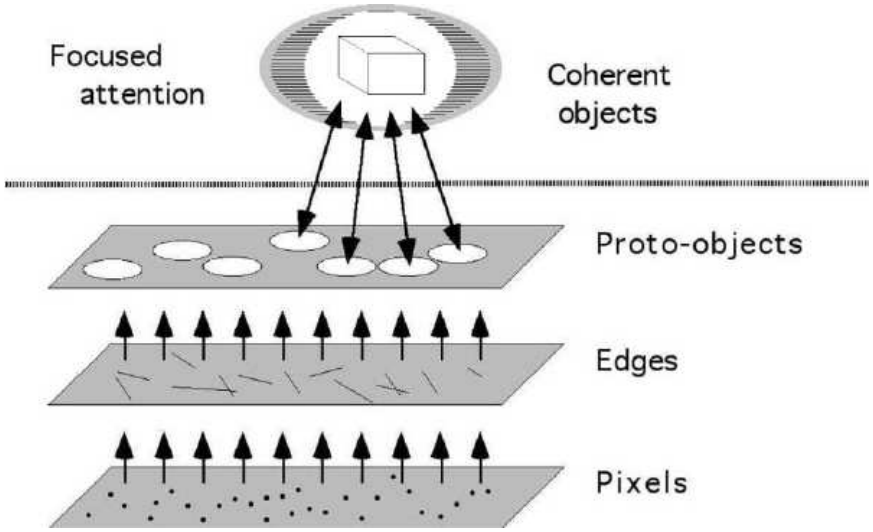


Figure 1.9: Proto-object. Based on the coherence theory by Rensink [8]. Proto-objects are preattentive, volatile structures with limited spatial and temporal coherence. They are computed rapidly and in parallel across the visual field. Attention is required to bind these proto-objects into the percept of a coherent object. Once attention is released, it dissolves back into its proto-object state. Image from [8].

percept of an object with a much a higher degree of coherence over space and time. Once attention is released, the object dissolves back to into its dynamic proto-object state. Furthermore, due to temporal continuity, any new stimulus at the location of the object is treated as a change to the existing structure rather than the appearance of a new one. Proto-objects can be better understood as the highest-level output of low-level vision and the lowest-level operand on which higher-level process can act, including attention.

There exists different hypothesis to how these proto-objects are actually computed within the human visual system. One hypothesis is described by Walter and Koch [60]. It suggests that proto-objects are a function of the saliency map. It assumes proto-objects are computed via a spreading activation within a small neighborhood of salient regions of the visual stimuli. Another hypothesis is based on the work of Craft et al. [20] which suggests that border-ownership neurons discovered in V2 [19] communicate with grouping cells in higher levels of cortex via white matter projects through high conduction velocity myelinated fibers. These grouping cells integrate border-ownership activity in an annular fashion giving rise to proto-objects. These account for the fast responses and computation of proto-objects in early stages of visual processing. The same normalization explained in feature-based saliency gives is computed as a function of these proto-objects. This gives rise to a proto-object based visual saliency map. Considering the promising results the later hypothesis (computation of proto-objects via border-ownership and grouping cells), we support

this hypothesis in the remaining work presented. Details of the biological mechanisms of this hypothesis for computation of proto-objects will be explained later in this thesis. Fig. 1.10 depicts the ability of the proto-object based saliency model to form the percept of an object even when it lacks complete closure but is perceived of an object ([1], [9].

1.3.2.3 Winner-Take-All (WTA) and Inhibition-of-Return (IOR)

After computing visual saliency, a mechanism known as winner-take-all (WTA) is applied [52]. The task of the WTA network is to extract the most conspicuous region of the visual field. It is this region that the focus of attention (FOA) is placed and is forwarded to further layers of visual processing. Another mechanism known as inhibition of return (IOR) works to inhibit the current region of FOA over time [61]. This causes the saliency at this current region to decrease overtime causing the next salient location to receive the FOA. This results in the shifting of FOA from one salient region to the next. It has also been suggested that the decision of the next location to attend to may not only be a function of magnitude of saliency, but also a function of properties such as proximity and similarity [52].

1.3.3 Motion Processing

Motion is a naturally occurring phenomena that plays an important role in both human and computer visual processing, and specifically visual attention. It has been

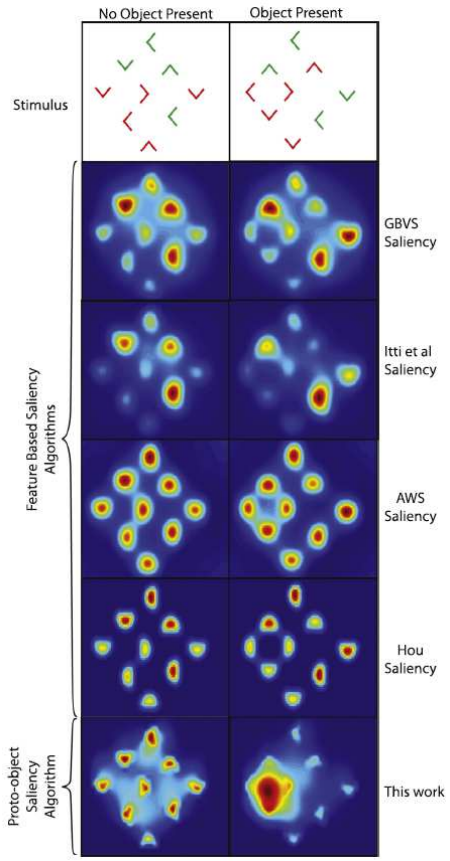


Figure 1.10: Saliency map output given 9 red and green 'L'-shaped elements arranged in various manners. These elements were used by Kimchi et al. [9] and subjects were tasked with identifying the color of a target element. Reaction times were fastest when the target formed part of an object. Therefore, attention is automatically drawn to the location of an object and therefore, a visual saliency model should be able to detect the region as a complete object. The proto-object based saliency model was capable of this while other SOTA models were not. Image from [1].

shown that given a dynamic visual stimulus, motion plays a more important role in the computation of visual saliency than other low-level features [62]. Considering our interest in bottom-up, feed-forward visual saliency, we are interested in how motion is computed early stages of visual processing.

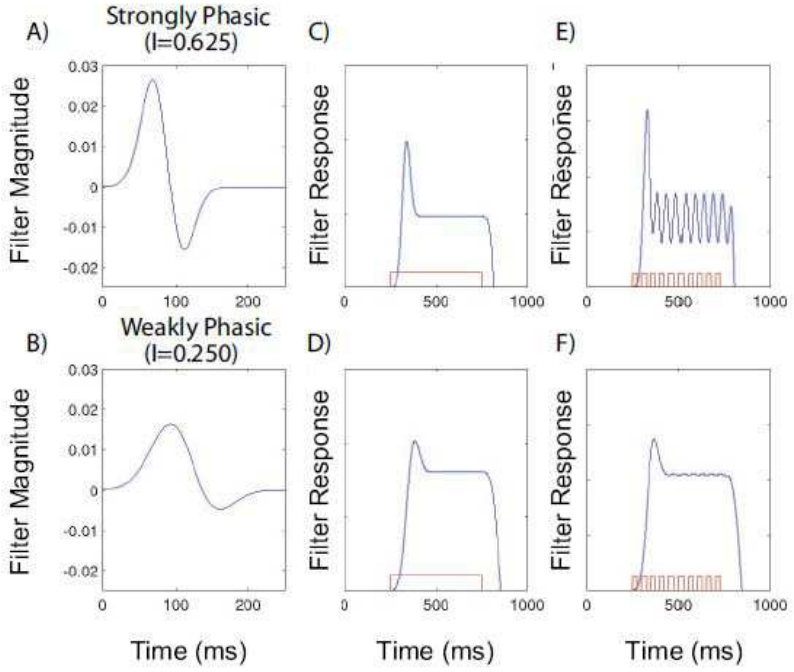


Figure 1.11: Plot A and B show the transfer function of a strongly phasic and weakly phasic filter, respectively. Plot C and D show the filter response to a constant stimulus onset, respectively. Plot E and F show the filter response to flicker motion (continuous onset/offset change). Strongly phasic filters are more sensitive to temporal change. Image from [10].

Henceforth, they require motion to be computed at early stages of visual processing. Neurophysiological research has shown that motion extraction occurs along

the dorsal pathway beginning in V1 and proceeds to middle temporal area (MT) and then continues to the medial superior temporal area (MST) [63]. Motion extraction in V1 can be represented by local spatiotemporal filters and show preference to spatial frequency, spatial phase, spatial orientation, and direction of motion. These are properties of the receptive field of simple cells in V1. Later stages of motion are responsible for computing velocity and optical flow. However, we are interested in preattentive motion, and therefore, consider motion extraction in V1 only. More specifically, we consider the receptive fields of non-direction selective simple cells in these models. There exists both strongly phasic and weakly phasic receptive fields of non-direction selective simple cells in V1 [10]. Roughly 20-25% of these cells are strongly phasic, the others weakly phasic. Strongly phasic simple cells exist within the magnocellular pathway and have high temporal resolution, high contrast sensitivity and low color sensitivity. Strongly phasic receptive fields have a shorter latency and have a stronger response to temporal changes. Weakly phasic receptive fields have a much slower and weaker response to temporal change (See Fig. 1.11).

These aspects of motion processing within early stages of visual processing are used throughout this doctoral work with respect to the implementation of a dynamic visual saliency model.

1.4 Neuromorphic Engineering

1.4.1 Origin and Motivation

In the field of neuromorphic engineering, we seek to engineer systems inspired by neuroscience. As previously noted, the brain is by far the most computational complex and efficient system operating under low-power, small-size, and light-weight constraints. By designing systems with physical characteristics and functionality that mimic the brain, we move closer to designing the most ideal systems. The quote below by Calimera et al. [11] summarizes the motivation behind neuromorphic engineering.

> *Understanding how the brain manages billions of processing units connected via kilometers of fibers and trillions of synapses, while consuming a few tens of Watts could provide the key to a completely new category of hardware (neuromorphic computing systems). In order to achieve this, a paradigm shift for computing as a whole is needed, which will see it moving away from current bit precise computing models and towards new techniques that exploit the stochastic behavior of simple, reliable, very fast, low-power computing devices embedded in intensely recursive architectures.* [11]

Neuromorphic engineering is a field coined by Carver Meade in the late 1980s [64]. In its originality, it is based on the notion of designing systems using Very Large Scale Integration (VLSI) analog technology for mimicking neural structures in the brain. In one of his original papers [64], he shows mathematically that (in 1990), with respect to power dissipation per operation, the brain is 10 million times more efficient than the best digital technology that we can imagine. The brain operates on different principles than those that are familiar to engineers. Biological solutions can

handle ill-conditioned, stochastic input streams. The use of adaptive analog VLSI elements allows us to design systems which can handle similar input data. We seek to understand the computational primitives of the brain and how they are used for computation and apply these paradigms to the electronic systems we build.

The approach to neuromorphic has two perspectives [11] which work together to solve the same problem. The first, and original view is the use of existing technologies for emulating neural mechanisms and characteristics. In this perspective, neuromorphic engineering can be seen as a means for simulating the brain at a similar speed, size, and power efficiency. This can be seen by the blue arrow in Fig. 1.12. The second perspective is engineering computing systems that have architectures and functionality that mimic the brain. This helps to overcome limitations of current technologies by making better use of the technology by having more power and size efficient systems. This can be seen by the white arrow in Fig. 1.12. In this doctoral work, we focus on the second perspective.

Meade demonstrated that rather than using transistors as on-off switches, we can exploit the analog properties of a transistor to model neural structures. Using current equations, he revealed the similarities between current flow in transistors and current flow in ion channels of neurons. One of the earliest neuromorphic circuits was the silicon neuron designed by Mahowald and Douglas in 1991 [65]. It modeled the passive ionic channels of a biological neuron using analog circuits. It further modeled leakage current and generation of action potentials (spikes). Most current

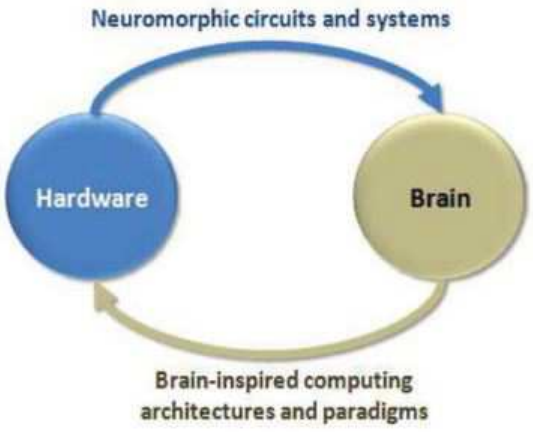


Figure 1.12: Diagram showing two different perspectives on neuromorphic engineering. The first (blue arrow), relates neuromorphic engineering to using existing hardware to emulate the brain. The second (white arrow), is the idea of using ideas of the brain to build more efficient computing systems. Image from [11].

neuromorphic systems build on the types of circuits introduced by Mahowald and Douglas.

From a high-level view, a typical biological neuron can be seen in Fig. 1.13. A single neuron is made of the following core elements: axon, dendrites, soma/cell body (including cell membrane), and nucleus (within soma). Neurons communicate via actional potentials (or spikes). Information is represented over time (spike rates) rather than digital bits. Neurons receive action potentials from other neurons via synaptic connections at the dendrites. This causes current to flow in and out of the cell membrane. These spikes can cause excitatory postsynaptic potentials (EPSP), which causes depolarization of the postsynaptic neuron's membrane potential (excita-

tory spike). This results in positively charged ions to flow into the cell, increasing the voltage across the cell membrane. On the contrary, inhibitory postsynaptic potentials (IPSP), causes positively charged ions to flow out of the cell, decreasing the voltage across the cell membrane (inhibitory spike). When the voltage across the cell membrane reaches a threshold, the soma outputs an action potential via its axon which has synaptic connections to the dendrites of other neurons. These components and functionality of the neuron are what is modeled in VLSI analog circuits. Other early neuromorphic systems included the silicon retina and cochlea. The silicon retina, for example, modeled photoreceptors and early stages of visual processing in analog VLSI circuits [66]. Similarly, the silicon cochlea was designed to model sensory and early stages of auditory processing [67].

1.4.2 Address Event Representation

Regardless of its application, most neuromorphic systems utilize address event representation (AER) protocol for representing information and computing in the time domain. This is a communication protocol developed by Mahowald in 1994 [13]. It is an asynchronous protocol using handshaking for receiving/transmitting events (spikes) to and from neuromorphic devices. This protocol uses time-division multiplexing (TDM). Address-events (AE) are transmitted along a shared digital bus. Each incoming address-event consists of an address corresponding to the address of the spiking computational element (e.g. neuron) that has generated a spike. The

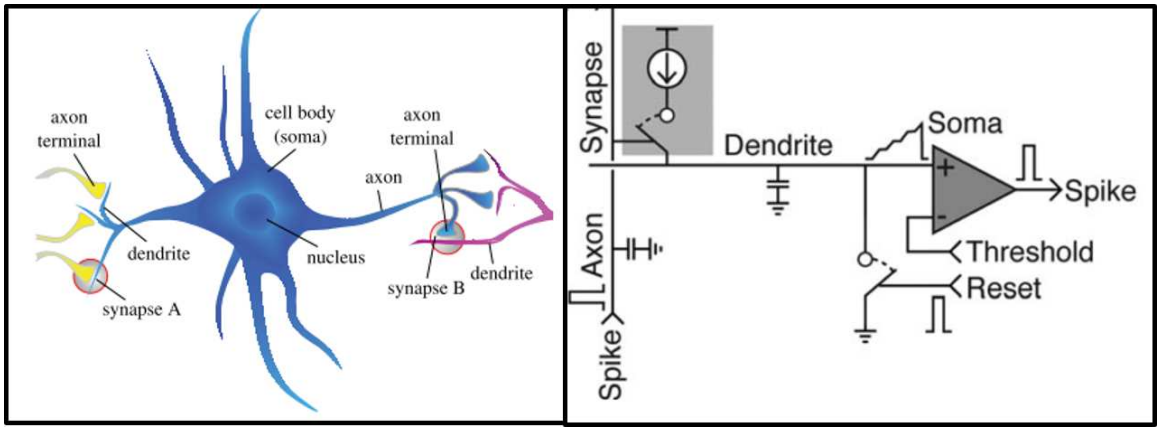


Figure 1.13: On the left is an image of a typical biological neuron. Each neuron consists of dendrides with synaptic connections to axons of other neurons (Synapse A/B). It also consists of a soma (cell body), with a cell membrane. Input signals are received via dendrites and output signals (action potentials/spikes) are transmitted to the dendrites of other neurons via the axon. These neuron components have circuit-based counterparts as seen by the example neuron circuit on the right. Each component of the neuron is modeled via VLSI circuits. The goal of neuromorphic engineering is to design systems such as these which have characteristics and functionality as the brain. Image from [12].

address of the AE is explicitly represented by the address on the digital bus. The time of the AE is implicitly represented by the time at which it is occurs and is received by the receiver of the neuromorphic system. In the same manner, as spiking elements generate spikes, AEs are outputted typically using a transmitter. A 4-phase handshaking protocol is used to interfacing between neuromorphic systems or interfacing with any external system. This handshaking is asynchronous and uses two additional signals, request (REQ) and acknowledge (ACK). A visual diagram of the AER protocol can be seen in Fig. 1.14. AER is ideal because rather than having N wires for each synaptic connection from one element to another, AER only requires $log_2 N$ wires.

- Phase 1: The address is latched onto the data bus and the REQ signal is asserted from the transmitter/sender.
- Phase 2: After the address is received, an ACK signal is asserted from the receiver, acknowledging that the address has been received.
- Phase 3: After the ACK signal is received by the sender, the REQ signal is deasserted.
- Phase 4: After deasserting the REQ signal, the ACK signal is deasserted.

In this section we introduce the original AER architecture. However, a more novel AER protocol using delay-insensitive asynchronous circuits has been presented by Lin et al [68]. This architecture decreases delay by using one-hot encoding and removing

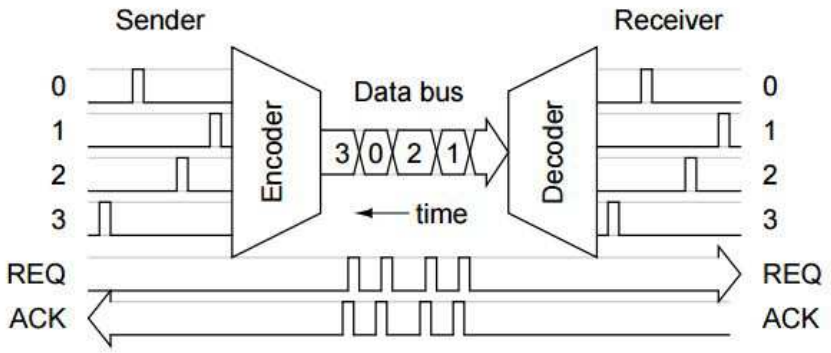


Figure 1.14: Visual diagram of the address event representation (AER) communication protocol [13]. It is an asynchronous communication protocol in which address-events are sent via a shared data bus using time-division multiplexing. A 4-phase handshaking protocol is used via the REQ and ACK signals for receiving/transmitting events. Image from [14].

the need for a *REQ* signal. We take advantage of this novel architecture in this doctoral work and further detail will be explained in later sections.

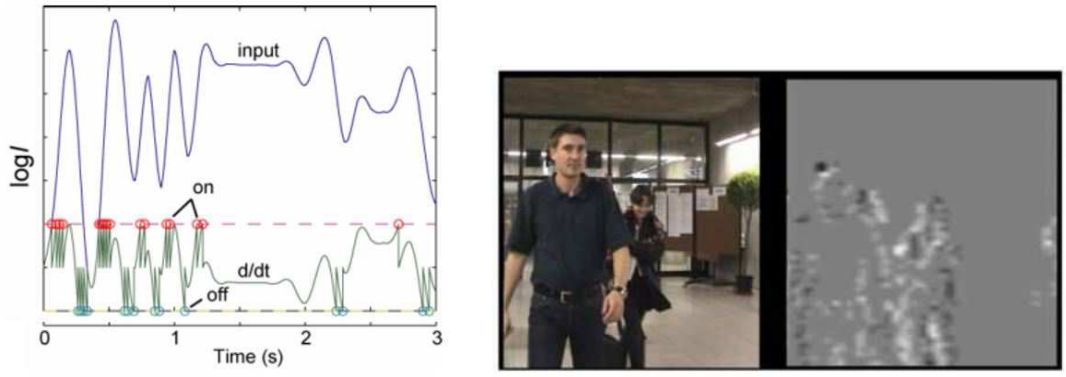
Neuromorphic engineering has evolved over the past few decades to model various sensory modalities and biological processing. In this work, however, the focus is on vision. In the proceeding sections we give a brief introduction to the visual processing aspect of neuromorphic engineering.

1.4.3 Neuromorphic Image Sensors

As noted in the prior section, one of the earliest systems designed within the neuromorphic engineering field was the silicon retina [66]. This system, designed in VLSI technology, emulated physical characteristics and functionality of photorecep-

tors (rods and cones) in the retina. It further modeled bipolar cells in early stages of visual processing. Since this silicon retina, many systems have either built on this design in building more advanced silicon retina, or neuromorphic systems have been designed to do event-based visual processing (post-retinal processing).

Since the original silicon retina by Mahowald and Meade, there have been several advancements in the silicon retina design including [69–74]. Silicon retina can also be referred to as event-based imagers or event-based cameras. The idea behind these biomorphic sensors is to convert light energy in the form of event streams mimicking photoreceptors seen in the retina. Most recently, two event-based cameras have, arguably, dominated the current research in the neuromorphic visual processing field. The first, in its originality, is called the Dynamic Vision Sensor (DVS) designed by [15]. The DVS is analogous to the magnocellular pathway as it is sensitive to temporal changes and less sensitive to spatial changes. Unlike traditional frame-based CMOS images, each pixel responds only to temporal changes in log intensity (See Fig. 1.15). Henceforth, static scenes to not generate any output. Output events (using AER protocol) are outputted when there exists relative changes in image intensity. Rather than sending redundant information from static regions of the image, only changes above or below a variable threshold is outputted. This results in ON- and OFF-events. ON-events represent a positive relative change in log intensity. OFF-events represent negative changes in log intensity. This results in images such as that seen in Fig. 1.15.



(A) (B)

Figure 1.15: DVS imager. Image A shows how ON-/OFF-events are generated when the relative log intensity changes (positively or negatively) by some threshold. If this threshold is reached, an event is generated. Image B shows a screenshot of video input and the DVS output. The white pixels represent an ON-event and black pixels represent OFF-events. Gray pixels represent no output (no change). In this video, the woman on the right is moving to the right and the man on the left is moving to the left. Image from [15].

A second event-based imager is the Asynchronous Time-based Image Sensor (ATIS) designed by Posch et al. [16]. Each pixel within the ATIS consists of a change detector circuit which senses relative log intensity changes in illumination similarly to the DVS. It outputs an event when it senses a change. The ATIS pixels are unique in that they also generate a second and third event representing the time to a low and high threshold-crossing of a photocurrent integration at the pixel which exhibited the log intensity change. The inter-spike interval (ISI) between these two spikes/events is inversely proportional to the pixel intensity. The output of the ATIS can be seen in Fig. 1.16. The ATIS utilizes an address event representation (AER) communication protocol. The ATIS models not only the magnoceullar ("where") pathway by detecting change in log intensity, it also models the parvocelluar ("what") pathway by outputting two additional events with an (ISI) inversely proportional to the pixel intensity at the pixel that experienced the change (spatial details). Such an event-based approach to imager design poses many advantages. These include low data throughput, no temporal resolution limitation in regards to frame-rate, high dynamic range, low power consumption and small area consumption.

1.4.4 Neuromorphic Processors

In this doctoral work, the focus is on the design of neuromorphic visual processors which operate on event-based input. The neuromorphic processors we focus on are those utilized for visual processing. We focus on hardware implementations on

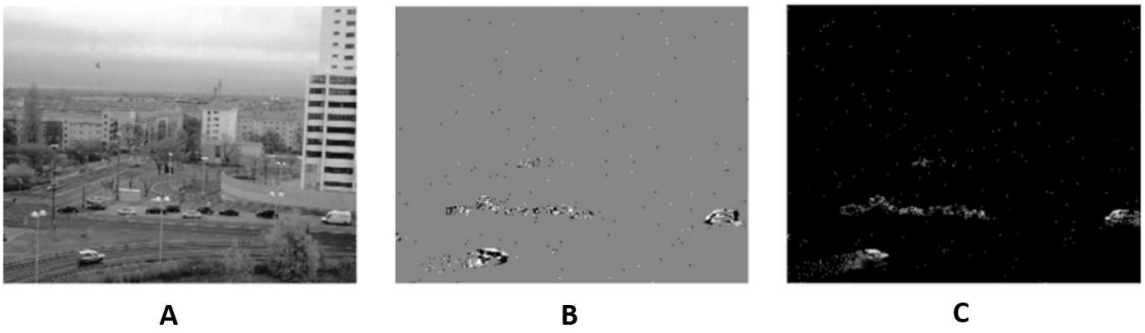


Figure 1.16: Results from ATIS output. Image on the left (A) is a screenshot of the raw video input. Image in the middle (B) is ON/OFF-event output detecting log intensity change. The image on the right (C) is the decoding of the pixel intensity from the ISI of the two additional events at the location of change. Image from [16].

both Field Programmable Gate Array (FPGA) and VLSI technology. There exists much research on neuromorphic designs on FPGA hardware. Orchard et al. [75] implemented the biologically-inspired HMAX (Heirarchical model and X) model for an object recognition task. Farabet et al. also developed a platform called "NeuFlow" which allows for design in Lua scripting language and synthesis onto FPGA for convolutional neural network applications [76]. These are only a few of many neuromorphic FPGA designs. There has also been research in the design of neuromorphic processors in VLSI technology. Current state-of-the-art neuromorphic systems are neural arrays including Neurogrid (array of analog neurons) [12], SpiNNaker (digital processors for event-based processing/communication) [28], BrainScales (array of analog neurons) [26], and TrueNorth (array of digital neurons) [27]. Each of these arrays have benefits and disadvantages in their design which will be discussed in more detail

in Chapter 5;

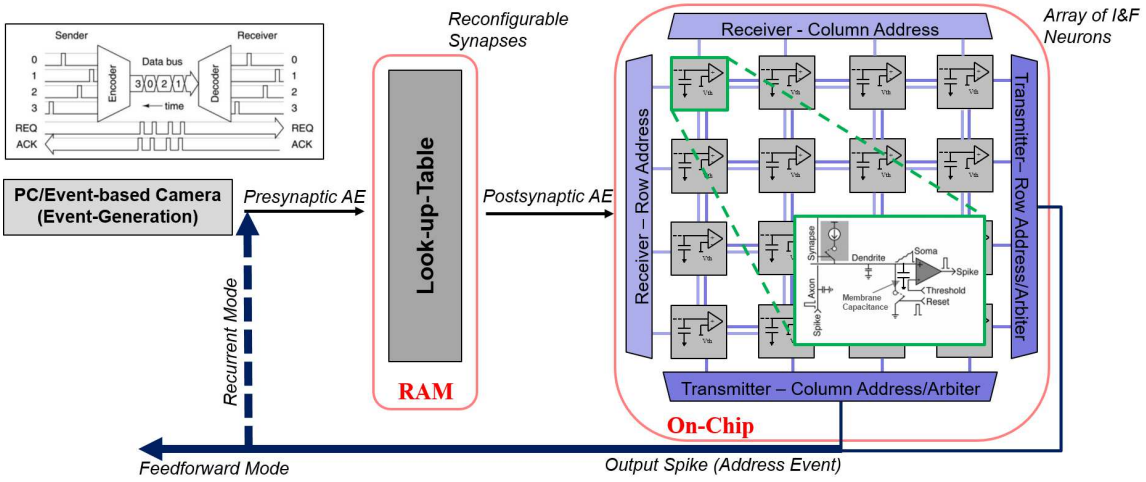


Figure 1.17: Diagram of the Integrate-and-Fire Array Transciever (IFAT). Presynaptic events go through a LUT which holds postsynaptic connections. These postsynaptic events are sent to the array of integrate-and-fire neurons. As neurons reach their threshold and generate events, the events are outputted via the AER-based transmitter.

The focus of this doctoral work is based on the FPGA emulation and VLSI design of the Integrate-and-Fire Array Transceiver (IFAT) [14, 77–79]. The IFAT is an array of analog neurons with dynamic, reconfigurable synapses. It uses AER communication protocol for sending/receiving events. Synaptic connections and weights are stored in a look-up-table (LUT) off-chip which allows for infinite number of synaptic connections. We utilize the IFAT system and enhance its design in building a neural array that can perform visual processing tasks. A diagram of the IFAT can be seen in Fig. 1.17.

1.5 Visual Pre-processing: Filtering, Dewarping, Visual Saliency

Visual pre-processing is essential for any autonomous mobile agent (i.e. micro aerial vehicles, MAVs, or unmanned aerial vehicles, UAVs) which must perform visual tasks such as object recognition, object tracking, or navigation in real-time. We focus on three primary areas of pre-processing:

- Image filtering (exhibited by neurons in early stages of visual processing)
- Image dewarping (compensate for ego-motion exists)
- Dynamic visual saliency computation (more efficient processing)

1.5.1 Image Filtering

With regards to the human visual system, pre-processing is analogous to the processes (filtering) which occur by neurons at early stages of visual processing. These filters are typically center-surround, low-pass, or edge receptive fields of ganglion cells in retina or simple cells in V1. Similarly, in computer vision, the raw output of camera is usually corrupted by noise, variations in illumination, or insufficient contrast. Therefore, filtering must be performed at the front-end. For a linear, invariant system, the input can be seen as an impulse, $\delta(x, y)$. The output response to this impulse is invariant to spatial location (space invariant) and can be described

as $g(x, y)$ (See Fig. 1.18). If we assume an input image, $f(x, y)$, a linear space invariant system with an output impulse response $g(x, y)$, the resulting output image will be $h(x, y)$ (See Fig. 1.18) [17].

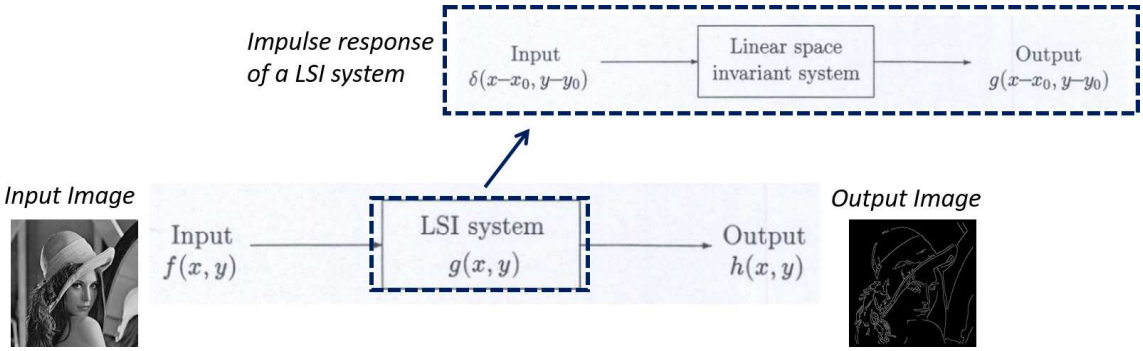


Figure 1.18: Image filtering. Input to a Linear Spatial Invariant (LSI) system can be seen as an impulse. Output of the LSI system is the impulse response. This LSI system can be a filtering process in which given an input image, for each pixel, an output response is computed, resulting in the output image. An edge filtering task is used as an example here.

Mathematically, the filtering process is represented as a convolution. Given an input image $f(x, y)$, an impulse response $g(x, y)$ (or filter), the output image is $h(x, y)$ (See Equation 1.1).

$$h(x, y) = f(x, y) * g(x, y) \tag{1.1}$$

The discrete convolution function can be seen in Equation 1.2.

$$h[i,j] = f[i,j] * g[i,j]$$
$$h[i,j] = \sum_{k=1}^{n}\sum_{l=1}^{m} f[k,l] \times g[i-k, j-l] \tag{1.2}$$

where $[i, j]$ is a pixel coordinate, m is the number of rows of the kernel, and n is the number of columns of the kernel.

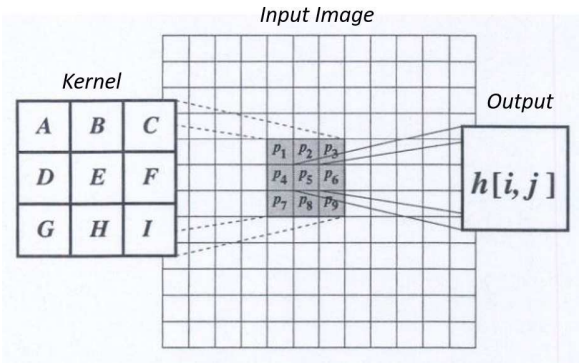


Figure 1.19: Diagram of convolving a filter/kernel over a single pixel. The kernel consists of coefficients A,B,C,D,...,I and a weighted sum with the corresponding 3 × 3 neighborhood results in $h[i, j]$ value at that pixel location in the output image. Image from [17].

The idea of convolving a filter/kernel over an image can be seen in Fig. 1.19. This idea is utilized throughout this doctoral work as we seek to design a neuromorphic system which can efficiently perform this convolution task.

1.5.2 Image Dewarping

Furthermore, when ego-motion (motion induced by motion of the image sensor, eyes or camera) exists, the raw information may be warped, and therefore, must be de-

warped or registered onto a single coordinate system to ensure accurate computation of higher-level visual tasks (e.g. object tracking). One can imagine a UAV performing an object recognition or tracking task in real-time. Camera motion is inherently present as the UAV navigates its environment. Henceforth, the camera output must be dewarped and registered onto a common coordinate system for efficient computation of these visual tasks. In this work, we focus on rigid 2-D warping/dewarping. Therefore, we assume rotation and translation motion only. Computer vision tasks typically use affine transformations for dewarping a pixel from one location to another (See Fig 1.20). An affine transformation assumes angle rotation (θ) and horizontal and vertical translation (t_h and t_v) is known.

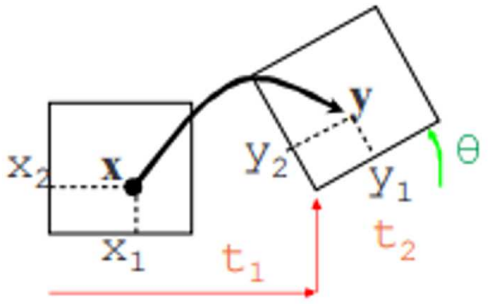


Figure 1.20: Diagram showing the dewarping operation. Pixel location at $(x1, x2)$ *is dewarped based on 2D rotation and translation to location* $(y1, y2)$.

Mathematically, the affine transformation of a mapping from (x_1, x_2) to (y_1, y_2) can be seen in Equation 1.3 and Equation 1.4.

$$y_1 = x_1 cos(\theta) - x_2 sin(\theta) + t_h \tag{1.3}$$

$$y_2 = x_1 sin(\theta) + x_2 cos(\theta) + t_v \tag{1.4}$$

where θ is rotation about a single 2-D axis and t_h and t_v is 2-D translation horizontally and vertically, respectively. This idea of dewarping is utilized throughout this doctoral work.

1.5.3 Visual Saliency: An Engineering Perspective

Lastly, as previously noted, visual saliency can serve as a pre-processing step for any visual system (including human visual system) for extracting only interesting regions of the scene for further processing. By using a visual saliency model at the front-end of any visual task, this can reduce data throughput and computation time of visual tasks. The biology behind visual saliency was explained in a previous section. Arguably, the most influential engineered model of visual saliency was presented by Itti et al. (1998) [18]. This model was based on the ideas of the Feature Integration Theory of Triesman and Gelade [47]. This biologically-plausible model has been used by engineers in both computer vision and computational neuroscience. The model overview can be seen in Fig 1.21.

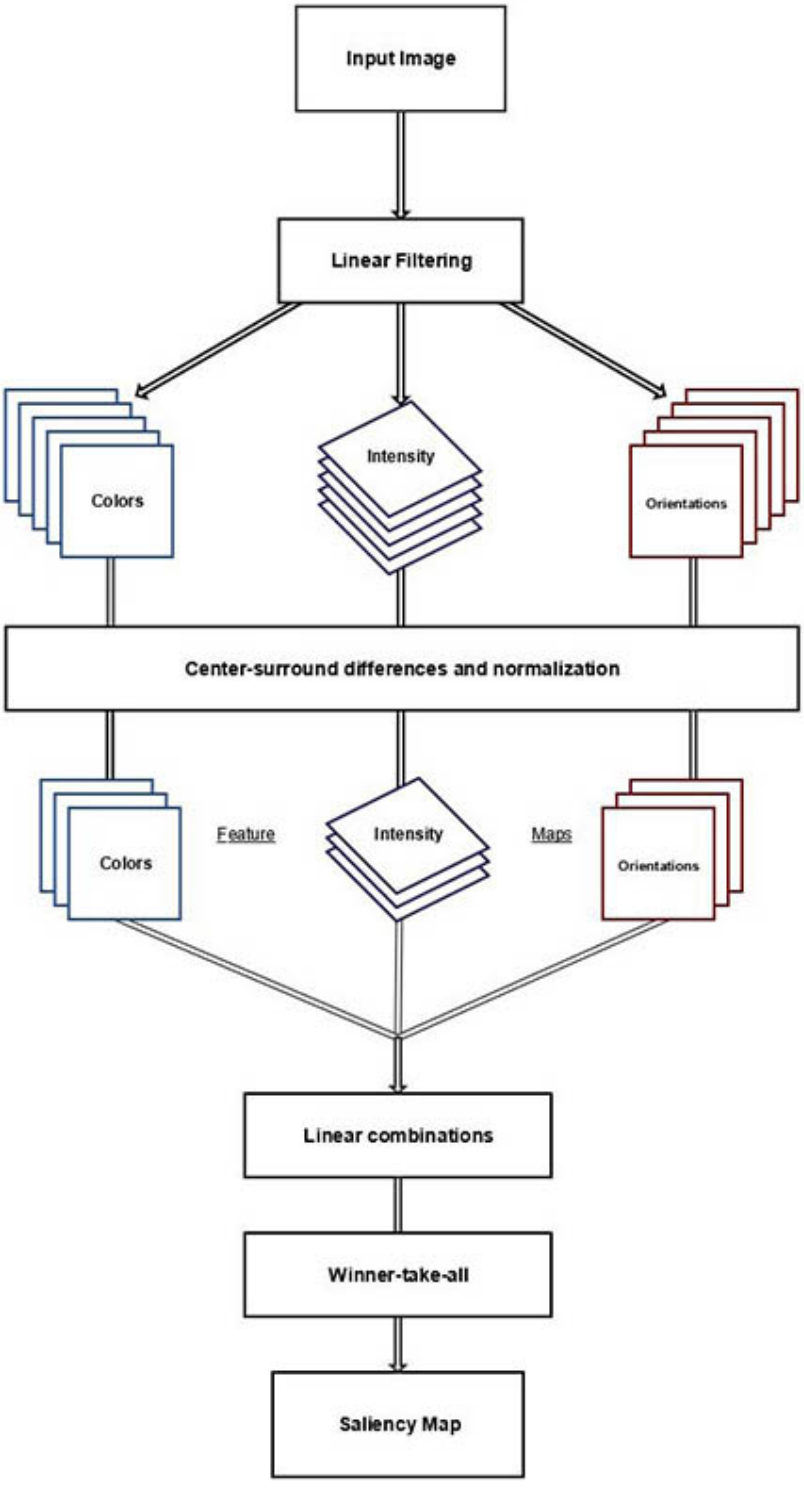


Figure 1.21: Diagram depicting Itti et al. model of visual saliency [18].

The model works by initially sub-dividing the visual RGB input into three individual feature channels: intensity, color, and orientation. Furthermore, computation is performed on an image pyramid within each of these feature channels. Center-surround operations combined with normalization operations work together to compute single conspicuity maps within each feature channel. The normalization operation works such that unique features are enhanced while common features are suppressed. This normalization operation, $N(.)$, works as follows:

1. Normalize values in the map to range $[0, M]$
2. Find location of global maximum, M
3. Compute average, $\overline{m}$, of all other local maxima
4. Globally multiply map by $(M - \overline{m})^2$

Finally, the conspicuity maps of each feature channel are linearly summed to form the final saliency map (Equation 1.5)

$$S = \frac{1}{3}(N(Intensity) + N(Color) + N(Orientation)) \tag{1.5}$$

where S is the saliency map. The saliency map is a single-valued map representing saliency within the visual field. Pixels with higher saliency are awarded higher values while those that are less salient have lower values. An example saliency map represented as a heat map can be seen in Fig 1.22. WTA and IOR is also applied

for shifting focus of attention. In this work we take advantage of some ideas of this model (e.g. the normalization operator), but rather focus more on the computation of dynamic proto-object-based saliency as we integrate motion into a proto-object-based model.

Figure 1.22: Example of the saliency map output of a visual saliency model represented as a heat map. In this case, the boat is most salient location.

1.6 Design Platforms

1.6.1 CPU - Central Processing Unit

CPU, or central processing unit, has been used throughout research for modeling designs prior to building them. Software such as MATLAB, Python, or C can be used to develop models. Most visual saliency models have been successfully implemented in software on CPU only [1, 18, 80]. While software design via CPU is more feasible with respect to implementation, it consists of overhead that causes slower processing speeds. Although there has been advancements in CPU design and can allow some parallel processing, it is difficult to take advantage of this parallel architecture. Fur-

thermore, it still contains overhead slowing down processing speed. Finally, CPUs are not ideal when the system must interface with the real world via autonomous mobile systems as they are power hungry and large in size. It is ideal for modeling and validation of designs which should later be implemented on application-specific hardware.

1.6.2 GPU - Graphics Processing Unit

GPUs, or graphics processing unit, is hardware designed for accelerating computer visual display. While CPUs consists of only few cores (high-end CPUs consists of 128 cores), GPUs contain thousands of cores allowing for more parallel processing. While processing speed increases for GPU implementations, GPUs development is more complex than for CPUs. There exists various visual saliency models implemented on GPUs [81, 82]. While real-time processing can be achieved via GPU design, the drawbacks are high power consumption and typically larger in size making them less feasible for applications with small-size and light-weight constraints (integration with MAVs).

1.6.3 FPGA - Field Programmable Gate Array

FPGAs, or field-programmable gate arrays, are very different than GPUs and CPUs. FPGAs are not processors in the same sense as GPUs and CPUs. Instead,

it consists of reconfigurable logic blocks and memory running on a clock. It consists of input and output signals which allow fast communication with other devices (e.g. USB, ethernet). FPGAs are one step below VLSI technology, in that the VLSI technology is simply an array of these digital logic and memory blocks which are fixed. However, HDL (Hardware Description Language), typically VHDL or Verilog, must be used to for development for FPGAs. This type of programming is very different from software development, in which, operations are executed serially. In HDL development, various blocks are implemented, synthesized and mapped onto the reconfigurable digital blocks for parallel operation. The advantages of FPGA design is that FPGAs are typically smaller in size, consume less power, and use reconfigurable logic blocks for parallel processing meaning there is a fast turnaround for testing and validating the implementation (in comparison to VLSI/ASIC design). They are more ideal for application-specific designs as it uses hardware more efficiently.

1.6.4 Very-Large-Scale-Integration

VLSI, or very-large-scale integration technology, also known as ASIC (Application-Specific Integrated Circuit), is the most ideal system for interfacing with real-world autonomous mobile systems. It allows for hardware/circuit design down to the transistor-level. It is ideal for designing application-specific hardware using both analog and digital components. It allows for low-power, small-size, and light-weight design making it ideal for a neuromorphic system. Systems designed in VLSI tech-

nology can be on the order of a few millimeters in size or even smaller. The drawback to VLSI design is turn-around time from design to receiving a fabricated chip can be on the order of a few months. Furthermore, design development for ASIC/VLSI chips may use HDL language and synthesize to transistor-level digital circuits, but also may require manual design of analog circuits.

1.6.5 Comparison Summary

In this work, we focus on the design of systems for implementation on FPGAs and ASICs. The Fig 1.23 shows a general ranking of these design platforms with respect to size/area, cost, speed, power, and turn-around time.

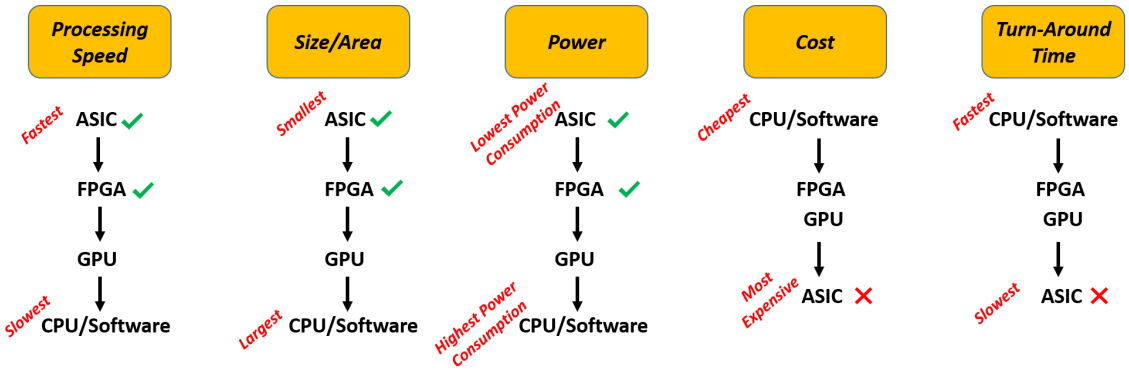


Figure 1.23: Ranking of design platforms with respect to various specifications. In this work, we focus on design on FPGA and ASIC.

1.7 Main Contributions

The following are the main contributions to the field of neuromorphic engineering as it pertains to vision and visual processing. Specifically, we focus on design of neuromorphic systems in hardware for performing tasks related to early stages of visual processing including dewarping, filtering, and dynamic bottom-up visual saliency.

1.7.1 Neuromorphic Systems – FPGA

1.7.1.1 Visual Pre-processing: Dynamic Visual Saliency

One prominent contribution (discussed in **Chapter 2**) is the ***design of a dynamic proto-object based visual saliency model*** by integrating motion into the Russel et al. model [1]. This novel dynamic visual saliency model allows us to consider temporal information when computing saliency. We show the importance of considering motion when computing saliency on dynamic natural scenes. Furthermore, we show that this model outperforms other current state-of-the-art dynamic visual saliency models in predicting human eye saccades on a given video dataset. This work was published in Biomedical Circuits and Systems in 2013 and presented in Rotterdam, Netherlands [83]. Extension of this work was then published at Conference on Information Systems and Sciences in Baltimore, MD, USA [84].

The recent advancements in virtual reality (VR) technology has enabled interesting

applications in various fields of research ranging from psychology to engineering. By immersing users into a 3-dimensional computer-simulated environment, VR allows for seemingly real interaction of virtual objects and space using specialized display screens. We present ***a modified visual saliency model for saliency computation in warped space*** for virtual reality applications as well as saliency computation on visual output from cameras with 360 degree, panoramic ("fish-eye") lens (discussed in **Chapter 2**).

Building on this work, the next contribution (discussed in **Chapter 3**) is the ***implementation of this real-time, dynamic proto-object based visual saliency model on FPGA hardware***. This is not only the first to our knowledge proto-object-based saliency model implemented on hardware, but it is also the first object-based visual saliency model implemented on hardware. We demonstrate how the model is comparable to the software-based model. This model is compared to other visual saliency models implemented on FPGA hardware. This system was presented and demoed at Biomedical Circuits and Systems in Atlanta, GA, USA in 2015 [22]. A journal publication is currently ...

1.7.1.2 Visual Pre-processing: Filtering, Dewarping

An additional contribution (discussed in **Chapter 4** is an ***FPGA implementation of the Integrate-and-Fire Array Transceiver (IFAT) in conjunction with stochastic computational elements for performing an event-based filtering (convolutions) and dewarping task simultaneously***. In this work, we assume camera motion is given and demonstrate its application for mobile agents which must perform visual tasks, specifically aerial vehicles. This work was sponsored by DARPA UPSIDE (Unconventional Processing of Signals for Intelligent Data Exploitation) project. This system served as an emulation of various unconventioal image processing tasks (e.g. dewarping, debayering) using event-based processing for WAMI (Wide Area Imagery) and robotics. This work was further published in Midwest Symposium on Circuits and Systems in 2015 and presented at Fort Collins, CO, USA [23].

Building on this FPGA implementation of the IFAT, also in **Chapter 4**, we discuss how we ***extended this system to take advantage of the silicon retina camera, the ATIS as well as a commercial IMU (Inertial Measurement Unit) in order to design a more complete event-based visual processing system on FPGA***. This is a front-to-back event-based system which takes advantage of the inherent stochasticity of the ATIS camera rather than imposing stochastic computational elements. The ATIS output serves as input to the FPGA-based IFAT

system which allows for a simultaneous dewarping and spreading task. The IMU, coupled to the ATIS, serves as input to the FPGA-based IFAT system enabling the dewarping task given this known camera motion. We demonstrate how this system can be integrated onto a drone (or any small, unmanned aerial vehicle) for performing the dewarping task. Such an event-based approach allows for low-power, light-weight, small-size design. This work was published and presented at a special session in International Symposium on Circuits and Systems at Montreal, Canada in 2016 [25].

1.7.2 Neuromorphic Systems – VLSI CMOS Technology

1.7.2.1 Generic Neuromorphic System

In regards to neuromorphic systems in VLSI technology, we have built on the work of Brandli et al. [85] and designed ***a working neural array of Mihalas-Niebur neurons in*** $0.5\mu m$ ***CMOS technology utilizing a single synapse and single cell body/comparator*** (discussed in **Chapter 5**). This neural array consumes considerably less power and area per neuron in comparison to other neural arrays designed in similar feature-size CMOS technology. This work was published and presented at International Symposium on Circuits and Systems at Baltimore, MD in 2017 [29].

In building event-based systems, especially large-scale neural arrays, typically an AER communication protocol is used. We present the ***a novel software for automating design of a state-of-the-art Delay-Insensitive AER Receiver and Transmitter in 55nm and 130nm CMOS VLSI technology*** (discussed in **Chapter 6**). Using a given design tool kit specific library of cells, this software automates the complete schematic, layout, and simulation schematics for both the AER Receiver and Transmitter. Using a hybrid MATLAB and SKILL script platform, the only input necessary is the number of rows, number of columns, the row pitch, and the column pitch and the system will automate the design.

1.7.2.2 Visual Pre-processing: Filtering, Dewarping

We demonstrate ***a novel 55nm array of 144 integrate-and-fire neurons using a conductance-based (switch-capacitor circuit) synapse and comparator (soma) within each neuron*** (discussed in **Chapter 6**). The novelty in this array is its design in 55nm CMOS technology allowing for low-power operation. Furthermore, this system uses the automated AER design previously discussed to demonstrate its proper functionality. We demonstrate this systems capability of performing dewarping and filtering tasks.

Finally, we ***utilize the Mihalas-Niebur neuron array for designing a complete board containing with peripheral circuitry and an FPGA interface***

for performing event-based image processing tasks (also discussed in **Chapter 5**). This board is a generic system for performing event-based tasks, in our case, visual tasks including filtering and dewarping. Such a system can be extended to multiple chips on a single board allowing for more complex parallel event-based processing utilizing more neurons. This work was also published and presented at International Symposium on Circuits and Systems at Baltimore, MD in 2017 [29].

1.8 Thesis Structure

For the remaining of this thesis, in Chapter 2 we discuss the implementation of the dynamic visual saliency model followed by the modified visual saliency model for computing saliency in warped space. In Chapter 3 we discuss the FPGA implementation of the dynamic visual saliency model. In Chapter 4, the FPGA emulation of the IFAT for event-based visual processing is discussed. Chapter 5 discusses the VLSI implementation of the IFAT using Mihalas-Niebur neurons. Chapter 6 will discuss the AER receiver and transmitter automation along with a small 144 neural array implemented in 55nm which utilizes this automated AER architecture. Finally, in Chapter 7 we conclude the thesis and discuss interesting future work and applications.

Chapter 2

Model: Dynamic Proto-object Based Visual Saliency

2.1 Overview

In the Introduction (Chapter 1), visual saliency was briefly discussed, both from a biologically perspective and engineering perspective. Visual saliency is essential for the human visual system as it allows us to process 100 Mbps [45] of spatial and temporal information per optic nerve. This is an overwhelming amount of information that is simply impossible for even the most sophisticated brain to process in parallel [46]. Henceforth, we instead attend to interesting regions of the visual scene and transmit only these interesting regions to further levels of visual processing. These interesting regions are better known as the salient regions of the visual scene, hence

visual saliency.

In this chapter we first discuss current state-of-the-art feature-based and object-based visual saliency models for static images. The details of the proto-object based visual saliency (POVS) model for static visual scenes will be discussed [1]. The primary objective of this chapter is to discuss in detail how we integrate a motion component into this POVS model, creating a novel dynamic proto-object based visual saliency (DPOVS) model which considers motion that may exist within the visual stimuli. We discuss two methodologies for integrating motion.

The first involves using motion to modulate border-ownership cell activity (DPOVS-BOM). The second involves using spatio-temporal receptive fields of neurons in V1/V2 as direct input to the visual saliency model (DPOVS-ST).

Finally, we discuss a modified proto-object based visual saliency model for computing visual saliency in warped space (POVS-WARP). Such a model has direct application to virtual reality and systems utilizing cameras with 360° ("fish-eye") lens, and can benefit from visual saliency computation as front-end processing to computing various visual tasks.

2.2 Background

2.2.1 Feature-based Visual Saliency - for Static Stimuli

Early models of visual saliency are feature-based and based on the Feature Integration Theory (FIT) [47]. The FIT is a two stage hypothesis explaining feature search and conjunction search. It states that in feature search, where objects are defined by a unique feature, occurs rapidly and in parallel. Conjunction search, where an object is defined by non-unique features, occurs serially and requires focused attention. Henceforth, feature-based saliency models are based on the notion that low-level features are computed pre-attentively, rapidly, and in parallel. Objects, defined by a combination of non-unique features, requires attention to bind the features into a single object. Koch and Ullman [52] extended this idea to explain how these independent feature maps give rise to bottom-up attention, forming a saliency map. This saliency map is a 2-D retinotopic map depicting the level of saliency at each pixel in visual scene.

Arguably, the most influential model of visual saliency (based on the ideas from Koch and Ullman) is the Itti et al. model (1998) [18]. This model can be seen in Fig. 2.1. It is a biologically-plausible, bottom-up, feature-based model of visual saliency. The model receives a static image as input. The image is decomposed into

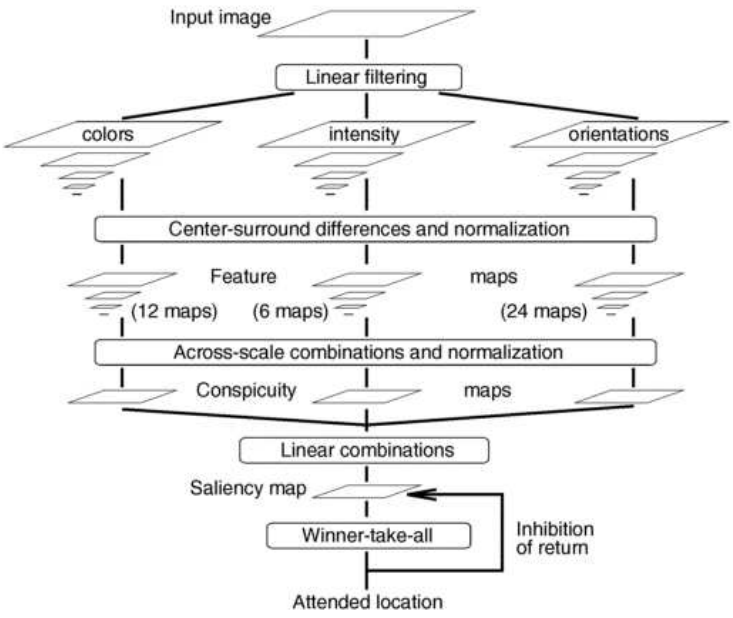


Figure 2.1: Diagram depicting Itti et al. model of visual saliency [18].

three feature maps: intensity, color, and orientation. Conspicuity maps are computed independently within each feature channel. Within each feature channel, an image pyramid is created to enable invariance to scale. A center-surround and normalization operation is applied within each feature channel, which gives high activity to unique features and low activity to common features. Within each feature channel, normalization is applied across the image pyramid and collapsed to a common level. A normalization operation is then applied across feature channels to enable invariance to modality and forming a conspicuity map for each feature channel. Finally, these maps are linearly summed across feature channels to form the final saliency map. This model was shown to predict human eye fixations better than chance. Many feature-based models have built on the Itti et al. model. There have been modifications in

the feature extraction as well as normalization method [86–90].

2.2.2 Object-based Visual Saliency - for Static Stimuli

Contrary to feature-based visual saliency models and to the ideas of the FIT, there exists saliency models supporting Gestalt psychology stating that the whole is perceived before the parts, or features [1, 60, 91, 92]. These models are referred to as object-based saliency models supporting the idea that attention does not depend on merely image features, but rather on the structural organization of the scene into perceptual objects. This approach to computing visual saliency is backed by neurophysiological and psychophysical evidence demonstrating objects are in fact perceived prior to features [59, 93, 94]. One hypothesis explaining object-based attention is the coherence theory introduced by Rensink [8]. The coherence theory states that there exists low-level proto-objects which are formed rapidly and in parallel across the visual field. These proto-objects are pre-attentive structures with limited spatial and temporal coherence. Focused attention is required to stabilize a small number of proto-objects, henceforth, generating the percept of and object with a much a higher degree of coherence over space and time. Once attention is released, the object dissolves back to into its dynamic proto-object state [8]. Furthermore, due to temporal continuity, any new stimulus at the location of the object is treated as a change to

the existing structure rather than the appearance of a new one. Proto-objects can be better understood as the highest-level output of low-level vision and the lowest-level operand on which higher-level process can act, including attention. A diagram of a proto-object can be seen in Fig. 1.9.

2.2.3 Proto-object Based Model - for Static Stimuli

In this thesis, we describe a dynamic proto-object based visual saliency model. This model (for video/dynamic stimuli) is a derivation of the original model of proto-object based saliency by Russel et al. [1] for static images. Therefore, it is important to discuss this model prior to discussing how motion is integrated into this model.

This model is an object-based, bottom-up, feed-forward model of visual saliency. It is based on the notion of proto-objects which may exist within the visual field. The model outperformed other state-of-the-art models ([18, 80] of visual saliency on predicting human eye fixations on a dataset of static images of natural scenes. A diagram of the model can be seen in Fig. 2.2. The model works as follows: it receives a color image (resolution of 640×480) and decomposes this image into three feature channels: intensity, color, and orientation. Within each of these feature channels are sub-channels. The intensity channel has one sub-channel. The color channel has four sub-channels: red-green opponency, green-red opponency, blue-yellow opponency, and

yellow-blue opponency. The orientation channel also has four sub-channels: objects oriented at 0°, 45°, 90°, and 135°. This totals to nine independent feature channels. Once the original color image is decomposed into these nine channels, within each channel, the feature map is successively down-sampled in steps of $\sqrt{2}$ to form an image pyramid spanning 5 octaves. Proto-object activity is then computed within each channel and at each level of the pyramid, independently. Proto-object activity gives rise to saliency with respect to figure-ground relationship within the visual scene. How the proto-objects (grouping activity) are computed will be discussed in Section 2.2.3.1. A normalization operation, N_1, is then applied to each grouping activity map to enhance maps with single proto-objects and suppress maps with multiple proto-objects. This normalization, N_1, works as follows:

1. The maximum, m, of the map being normalized is determined.

2. The average of the other local maxima, $\bar{m}$, is determined.

3. Normalization is applied by a global, element-wise multiplication of the map by $(m - \bar{m})^2$.

This normalization (N_1) is a function of the proto-object activity and the difference between the maximum activity and average maximum activity. Following this normalization, similar computation to that in the Itti et al. (1998) model [18] is performed. The image pyramids within each channel are collapsed by scaling each level to a common level/resolution and summing. This results in a single conspicuity

map within each channel. These nine conspicuity maps are then normalized using a second (but similar) normalization operator, N_2. The normalization, N_2, works as follows:

1. The map is normalized between a range $[0, ..., M]$.
2. The maximum, m, of the map being normalized is determined.
3. The average of the other local maxima, $\bar{m}$, is determined.
4. Normalization is applied by a global, element-wise multiplication of the map by $(m - \bar{m})^2$.

The only difference in this normalization is the additional first step of normalizing each map to a common range $[0, ..., M]$. This step is necessary for allowing invariance to modality (feature). Finally, these normalized conspicuity maps are linearly summed to form the final saliency map.

2.2.3.1 Grouping Mechanism

What is unique in this model is the computation of proto-objects and saliency with respect to proto-objects. The computation of proto-objects is based on biological evidence that border-ownership neurons communicate with grouping cells encoding for grouping activity (or proto-object activity). Neurons encoding border ownership (one-sided assignment of a border to a region perceived as a figure) have been discovered in early stages of visual processing, predominantly in V2, by Zhou et al. [19].

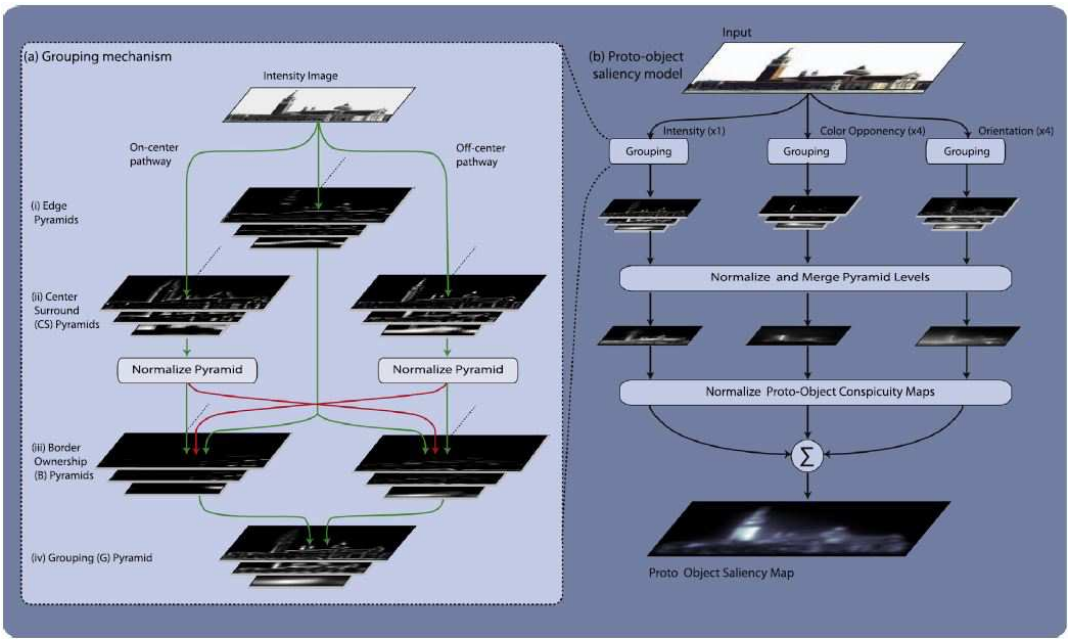


Figure 2.2: The original Russel et al. proto-object based saliency model [1]. The grouping mechanism can be seen on the left, (a). The complete model flow can be seen on the right, (b). Details of the model can be found in Section 2.2.3.

Results of this work can be seen in Fig. 2.3. Spiking activity was recorded from a single border-ownership cell in V2. As seen in Fig. 2.3, the stimulus used were squares of various sizes, both dark and light. The border-ownership cell only responded when the edge of the square object was within the receptive field of the cell. Furthermore, this response was also modulated by the side at which the object was placed. This border-ownership that was recorded from showed preferential to the object/square placed on the left-side. Regardless of size or color, spiking activity increased when the square was placed on the left-side. Originally, the work of Li [95] suggested that these border-ownership responses arise from lateral propagation between edge signals of a region deemed as figure. However, the work of Zhou et al. [19] noticed that the time of arrival of the border ownership signals was $\sim$20ms. Li's hypothesis is a slower process and does not explain the fast arrival of border ownership signals. Zhang et al. (2010) rather suggested, from neurophysiological evidence [96], that these border ownership neurons communicate with grouping cells via fast white matter projections (via myelinated fibers with high conduction velocity). The grouping neurons integrate object features into tentative proto-objects prior to recognizing the object. This grouping activity gives rise to perceptual organization of the visual scene into figure-ground relationship, hence, tentative proto-objects.

The modeling of this grouping mechanism is inspired by Craft et al. [20] and can be seen in Fig. 2.4. The first stage of this mechanism is extraction of object edges, similarly to the receptive field of simple cells in V1. Both odd (S_o) and even (S_e)

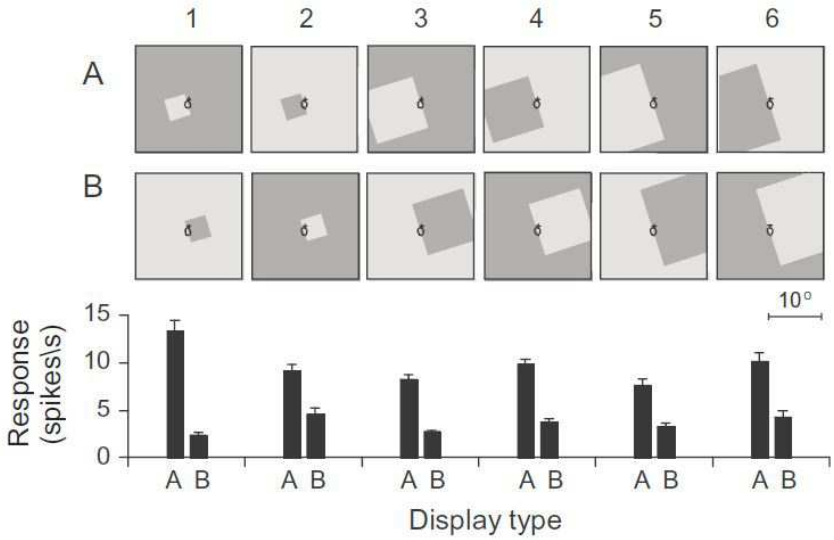


Figure 2.3: Spiking activity was recorded from a single border-ownership cell in V2. The stimulus used were squares of various sizes, both dark and light. The border-ownership cell only responded when the edge of the square object was within the receptive field of the cell (Rows A and B). Furthermore, this response was also modulated by the side at which the object was placed. This border-ownership that was recorded from showed preferential to the object/square placed on the left-side (Row A). Regardless of size or color, spiking activity increased when the square was placed on the left-side [19].

are combined to form complex cell responses (C). These complex cells are contrast-invariant edge responses which directly excite Border Ownership (B) neurons. The Border Ownership neurons representing right borders is B_0 and the Border Ownership neurons representing left borders is B_π. In order to extract information regarding to existence of objects, a center surround operation is performed (CS cells). This is similar to the receptive field of neurons found within the retina and LGN. Both ON- and OFF-center receptive fields. This is necessary for detecting dark objects on light backgrounds (CS_D neurons) as well as light objects on dark backgrounds (CS_L neurons). The border ownership responses to the complex cells are modulated by the center-surround cell responses. Excitation from the center-surround coding for figure on the Border Ownership cells preferred side increases Border Ownership activity and center-surround activity on the non-preferred side inhibits/suppresses border-ownership activity. Finally, Border Ownership activity is integrated in an annular fashion to give grouping cell activity (G cells). This grouping activity is representative of proto-object activity giving rise to figure-ground relationship of the visual scene.

2.3 Related Work

As previously discussed, early models of visual saliency models, both object and feature-based, compute saliency only on static visual stimuli. These models do not

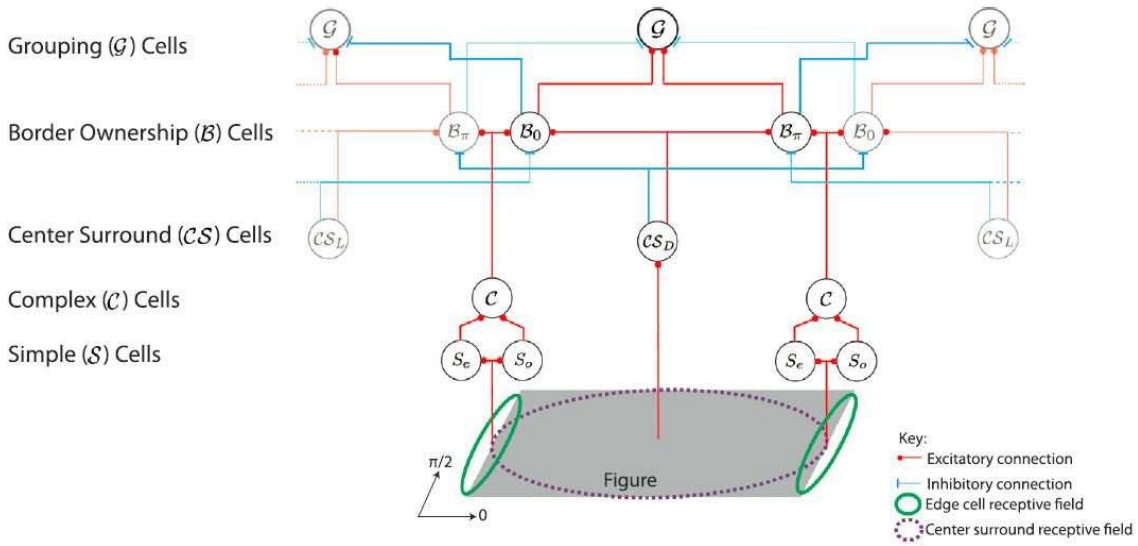


Figure 2.4: Visual representation of the grouping mechanism inspired by Craft et al. [20]. See text in Section 2.2.3.1 for further details. Note red lines represent excitatory connections while blue lines represent inhibitory connections. Furthermore, lines of strong contrast represent high activation and low contrast represent low activation.

consider motion that may exist within the visual scene. To validate these saliency models, datasets of images with attached human eye fixation data are used to compare how well the saliency model predicted these eye fixation data. However, the world we exist in is dynamic and constantly changing. Motion is a naturally occurring phenomena that plays an important role in both human and computer visual processing, and specifically visual attention. It has been shown that given a dynamic visual stimulus, motion plays a more important role in visual saliency than other low-level features [62]. Thus, it is important to consider the temporal dynamics of the visual stimuli when computing visual saliency. More recently, there have been saliency models, which do consider motion in developing a dynamic saliency map.

Rosenholtz introduced a simple saliency model for predicting motion popout phenomena [97]. This model suggests saliency as an outlier to a statistical distribution. It utilized the Mahalanobis distance between a given point and the mean of the distribution of velocities (or other feature values) to compute saliency. Although successfully predicted results of classical motion experiments, its computational mechanism do not have any biological correlates.

Gao et al. developed a model, which considers motion in a biologically-plausible manner using spatiotemporal Gabor filters [98]. This model uses KL divergence between distribution of pixel feature responses from the pixels local region. Similarly, Itti et al. extended their original model to include two additional feature channels aside from color, intensity, and orientation [21]. These are flicker (on-set/off-set) and

motion channels. Although this work noted that motion played the most important role in predicting human eye saccades on a video dataset, the model does not inherently suggest motion is more influential in predicting saliency than the other feature channels. Furthermore, both models support feature-based saliency although it has been shown that objects predict saliency better than features.

Seo et al. proposed a self-resemblance method for computing saliency [99]. It is a feature-based saliency model which considers features over space and time. Similarly to Gao [98], it also computes saliency by using statistics to measure likelihood of saliency at a given pixel relative to its local neighborhood. This model is feature-based and instead uses thresholding to compute proto-objects as a function of salient features, which is not supported by biology in regards to computing proto-objects.

Itti and Baldi introduced a saliency model which considers motion as an additional feature [100]. It does so in a similar manner as the Itti et al. model in that it includes flicker and motion as additional feature channel [21]. As with the other feature channels in this model (color, orientation, intensity), saliency is computed using Bayesian surprise. Similarly to Seo and Gao's work [99], it is based on statistics and uses Bayes theorem to statistically compute how much a new observation differs from its prior. Like in the Itti et. al model, this supports the idea that features are perceived prior to objects, although [59, 93, 94] suggest otherwise.

Zhang et al. computed saliency on dynamic scenes by utilizing separable spatiotemporal filters within each feature channel (color, orientation, and intensity) [101].

This approach is biologically-plausible in its computation however, not only is this model feature-based, but furthermore, it is computes saliency based on statistics and requires learning of the probability distribution for each feature.

Marat et al. developed a saliency model which computed a saliency map as a function of static features and then computes a second saliency map as a function of dynamic features [102]. These two saliency maps are fused for form the final saliency map. This fusion of static and dynamic saliency maps and method in which the dynamic saliency map is computed (using optical flow) does not show any biological correlates to low-level vision for bottom-up saliency.

Guo et al. presented a feature-based saliency model that uses the phase spectrum of the images Fourier transform to compute visual saliency, initially on static grayscale images [103]. However, more recently, they used the phase spectrum of the quaternion Fourier transform to compute saliency also on color and motion features. Although this method is inspired by biology in regards to the features it extracts, the method of using a quaternion Fourier transform representation to compute saliency was not shown to have biological correlates.

Finally, Liu et al [104]. presented a saliency model which considers both static and motion features. However, it uses learning to compute the salient regions of the image/video.

In the remaining of this chapter, two approaches for integrating motion will be discussed. Both are derivations of the proto-object based visual saliency model by

Russel et al. [1]. The first approach consists of modulating border-ownership activity with motion information (DPOVS-BOM). The second approach uses the motion information directly as input into the remaining of the visual saliency model, using separable spatiotempoteal filters (DPOVS-ST). Each of these models are bottom-up, object-based dynamic visual saliency model based on the notion of proto-objects. This model is biologically-plausible in that the temporal filters used are temporal filters observed in simple cells in V1 in both the parvoceullar and magnocelluar pathways. Furthermore, this model builds on the proto-object saliency model which supports the idea that objects are perceived prior to features, and henceforth, saliency is computed as a function of dynamic proto-objects existing within the visual field opposed to features only. Similarly to [1], these proto-objects are computed based on the notion of border ownership and grouping cells found in V1/V2 levels of visual cortex. The temporal filters used and how they are integrated into the proto-object based model will be discussed.

2.4 Representing Motion

In both models discussed in this paper, we seek to represent motion in our model in a biologically-plausible manner. These models are purely bottom-up and feed-forward. Henceforth, the focus in this work is motion computed at early stages of visual processing, and further, preattentive visual processing. Neurophysiological

research has shown that motion extraction occurs along the dorsal pathway beginning in V1 and proceeds to middle temporal area (MT) and then continues to the medial superior temporal area (MST) [63]. Motion extraction in V1 can be represented by local spatiotemporal filters and show preference to spatial frequency, spatial phase, spatial orientation, and direction of motion. Later stages of motion are responsible for computing velocity and optical flow. This motion processing requires attention [105]. However, we are interested in representing preattentive motion, and therefore, consider motion extraction in V1 only.

It should be noted that in this work we utilize the receptive field non-direction selective simple cells (although there exists direction-selective simple cells as well). As previously discussed, there are two pathways within the visual system: magnocelluar and parvocelluar pathway [63]. Each of these pathways begin have a select population of retinal cells, which then project to the LGN, and further project to primary visual cortical cells. Strongly phasic simple cells exist within the magnocellular pathway and have high temporal resolution, high contrast sensitivity and low color sensitivity. Cells in the parvocelluar pathway are weakly phasic and have low contrast sensitivity and high color sensitivity but low temporal resolution. Strongly phasic cells typically have a strong excitatory phase followed by a strong inhibitory phase. Weakly phasic cells typically have a strong excitatory phase followed by a weak inhibitory phase, resulting in a weaker response to motion. Approximately 20-25 percent of the population of non-direction selective simple cells in V1 are strongly phasic. The remaining are

weakly phasic. The temporal filters used in the models to be discussed are modeled to fit the receptive fields of these strongly and weakly phasic cells found in the primary visual cortex [63, 106].

2.4.1 Temporal Filtering

Before discussing the temporal filters used in this model, it is important to understand temporal filters and their responses. In this work we assume three dimensions: spatial dimensions along two-axis (x-axis and y-axis) and time along a third axis (t-axis). This is visually demonstrated in Fig. 2.5. In this diagram, Fig. 2.5A shows an edge moving from right to left over time (snapshot of this motion). The edge is spatially about the x- and y-axis. In Fig. 2.5B, the motion of the edge is depicted. The time axis (t) is the third dimension coming out of the page. Over time, the edge moves further to the left. To better visualize the effect of time and applying a temporal filter, we assume that the y-position is constant and the edge moves only about the x-axis. This allows us to represent this edge motion about two axes (x- and t-axis), as seen in Fig. 2.5C.

While spatial filters respond to stimuli/visual information spatially about the x- and y-axis, temporal filters' response is a function of the stimuli at the current time as well as at prior time s(for some specified duration). Fig. 2.6 shows the various positioning of the temporal filter over time using the two dimensional representation as in Fig. 2.5C (x vs. time). In Fig. 2.6A-C, the temporal filter used has an excitatory

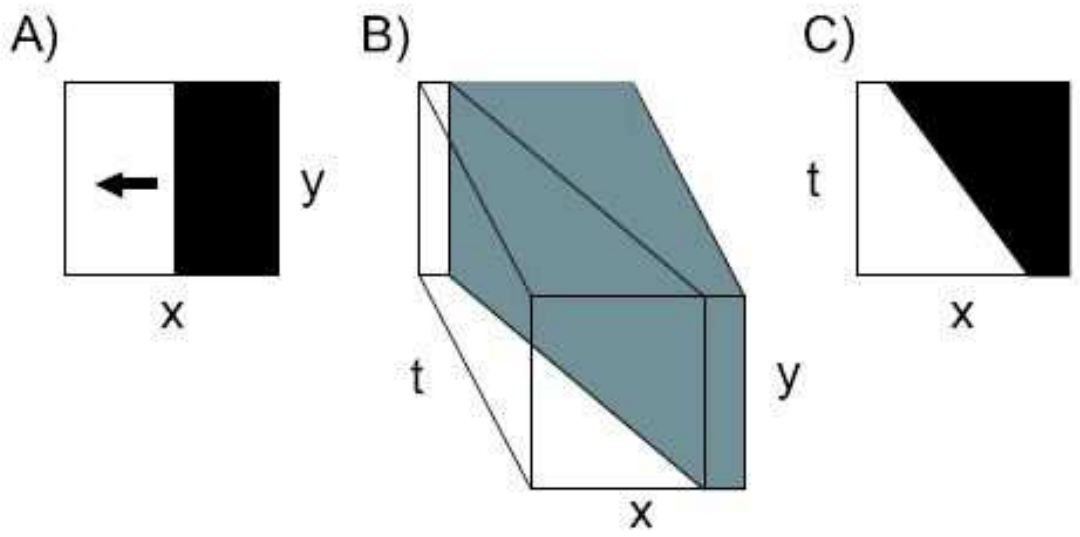


Figure 2.5: Visualization of three dimensions: x-axis and y-axis representing 2-D spatial dimensions and t-axis representing the temporal (time) dimension. Image A depicts an edge about the x and y axes moving from right to left over time. Image A depicts a snapshot of this motion. Image B visualizes the temporal axis and the motion from right to left. Image C depicts a 2-D representation of the temporal axis vs. the x-axis, assuming the y-dimension is constant. Image from [10].

component and inhibitory component. The output of the filter is a function of the current time and 150ms prior to the current time. Fig. 2.6A shows the temporal filter spanning over the first 150ms when the edge has not reached the center of the x-axis. Therefore, the response of the temporal filter will be zero since the excitatory component is equal to the inhibitory component. In Fig. 2.6B, the temporal filter is at 300ms. At this point in time, at the center of the x-axis, the edge is mostly in the excitatory region of the filter and less in the inhibitory region. This results in an overall positive response. Finally, in Fig. 2.6C, at 450ms in time, the edge has passed the center of the x-axis and the excitatory response is equal to the inhibitory response, resulting in a net zero response.

2.4.2 Biologically-plausible Temporal Filters

The work of Parkhurst [10] and De Valois et al. [63] was used to model the transfer function for a biologically-plausible temporal filter modeling the temporal receptive field of strongly phasic and weakly phasic, non-direction selective simple cells in V1. The approximation of the transfer function of the V1 simple cell temporal receptive field can be seen in Equation 2.1.

$$r(t) = \alpha(t - \tau - \delta)e^{\beta(t-\tau)^2} \tag{2.1}$$

- α - response amplitude parameter

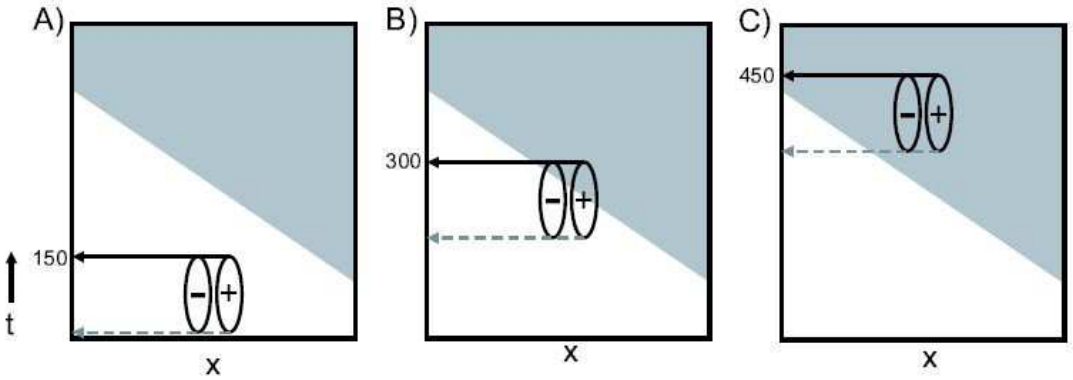


Figure 2.6: This image depicts various positioning of the temporal filter in time. The temporal filter has an excitatory component (+) and a negative component (-), which in total spans 150ms. In Image A, the temporal filter spans over the first 150ms when the edge has not yet reached the center of the x-axis. Therefore, the response of the temporal filter will be zero since the excitatory response is equal to the inhibitory response. Image from [10]. In Image B, the temporal filter is at 300ms. At this point in time, at the center of the x-axis, the edge is mostly in the excitatory region of the filter and less in the inhibitory region. This results in an overall positive response. Finally, in Image C, at 450ms in time, the edge has passed the center of the x-axis and the excitatory response is equal to the inhibitory response, resulting in a net zero response.

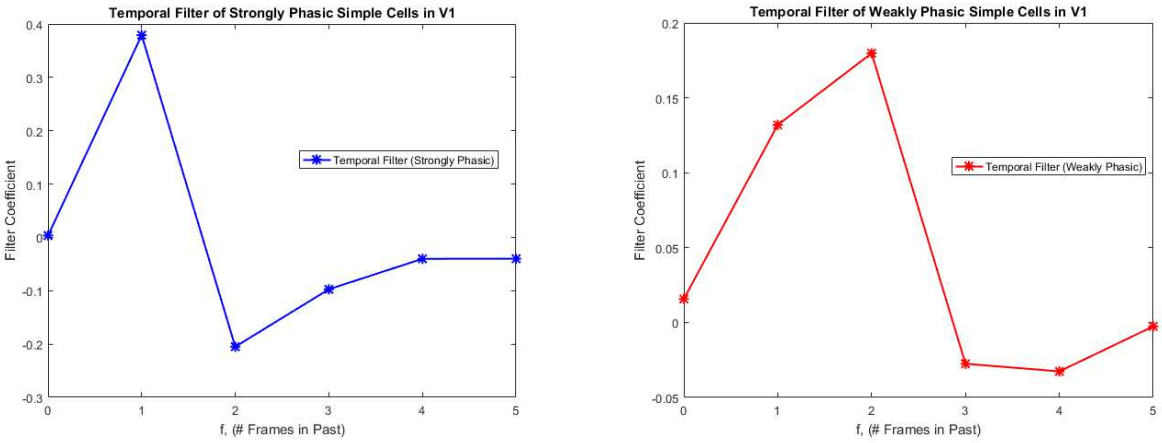
- β - response amplitude parameter
- τ - time shift parameter
- δ - determines degree to which weakly or strongly phasic in time

These parameters were fit to model the temporal response profile of strongly phasic and weakly phasic cells in V1 from neurophysiological recordings [10]. These parameter values can be seen in Table 2.1.

Table 2.1: Parameters for Strongly/Weakly Phasic V1 Simple Cell Temporal Response

Type	α	β	τ	δ
Strongly Phasic	-0.00161	-0.00111	86.2	5.6
Weakly Phasic	-0.000487	-0.000466	116	20

These temporal profiles for strongly and weakly phasic simple cell receptive fields are applied in our dynamic model of proto-object based visual saliency. The methodology in how they are applied will be discussed in proceeding sections. A visual representation of the filters can be seen in Fig. 2.7. As previously noted, the strongly phasic temporal filter in Fig. 2.7A, has a strong positive/excitatory lobe and a strong negative/inhibitory lobe. The weakly phasic temporal filter in Fig. 2.7B, has a strong positive/excitatory lobe and weak negative/inhibitory lobe. While the y-axis is the filter coefficient, the x-axis is the number of frames in the past based on 24 frames per second input image sequence.



(A) (B)

Figure 2.7: Visual plot of the strongly phasic (Plot A) and weakly phasic (Plot B) temporal filters representative of the receptive field of simple cells in V1. The x-axis is the number of frames of a video sequence in the past based on 24 frames per second input image sequence. The y-axis is the filter coefficient

Fig. 2.8 shows the response to various types of stimuli. The value I represents the degree to which the filter is strongly phasic, hence the strongly phasic filter has a larger value for I ($I = 0.625$) and the weakly phasic filter has a smaller value for I ($I = 0.250$). Fig. 2.8A and Fig. 2.8B are the strongly and phasic filters, respectively. Fig. 2.8 are their responses to

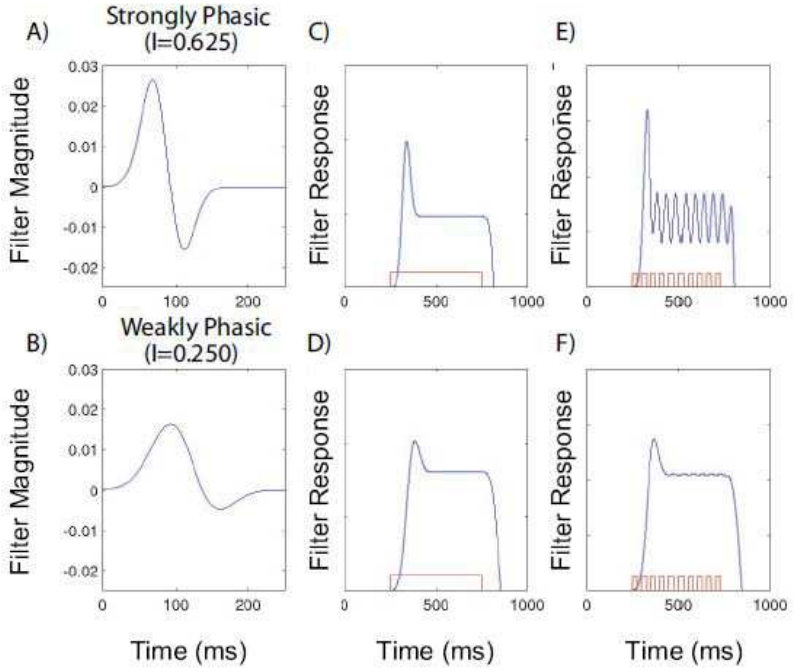


Figure 2.8: Plot A and B show the temporal profile of a strongly phasic and weakly phasic filter, respectively. Plot C and D show the filter response to an abrupt then constant stimulus onset, respectively. Plot E and F show the filter response to flicker motion (continuous onset/offset change). Strongly phasic filters are more sensitive to temporal change. I is a ratio representative of the degree to which the filter is strongly phasic. Image from [10].

It is important to note that each temporal filter has ideal biologically-plausible properties. These include that frames more than 250ms in the past to not contribute to the current saliency. If part of the visual stimulus does not change over an extended amount of time, there is a lower temporal response. And finally, these temporal filters in general have a stronger response to onset and offset of objects within the scene. These temporal dynamics are similar to those seen in the model of visual saliency depicted in [101], however, their model incorporates learning. In the following sections we will discuss the complete dynamic proto-object based visual saliency model using two approaches. In each approach, these biologically-plausible filters are utilized for integrating the motion component into the models.

2.5 Model 1 - DPOVS-ST (Space-time Separable Filters)

The first model for dynamic proto-object based visual saliency is the DPOVS-ST (Dynamic Proto-Object based Visual Saliency - SpatioTemporal filters). It utilizes the idea of separable space-time filters in integrating motion. This biologically-plausible model is based on the idea that simple cells in the magnocelluar and parvocellular pathways act as spatiotemporal filters. They not only extract spatial information preattentively, but also temporal information. This model is based on the original proto-object based visual saliency model by Russel et al. [1], however, now extracts

temporal and spatial information for computing saliency on dynamic scenes where motion exists in the visual stimuli. The complete model can be seen in Fig. 2.9.

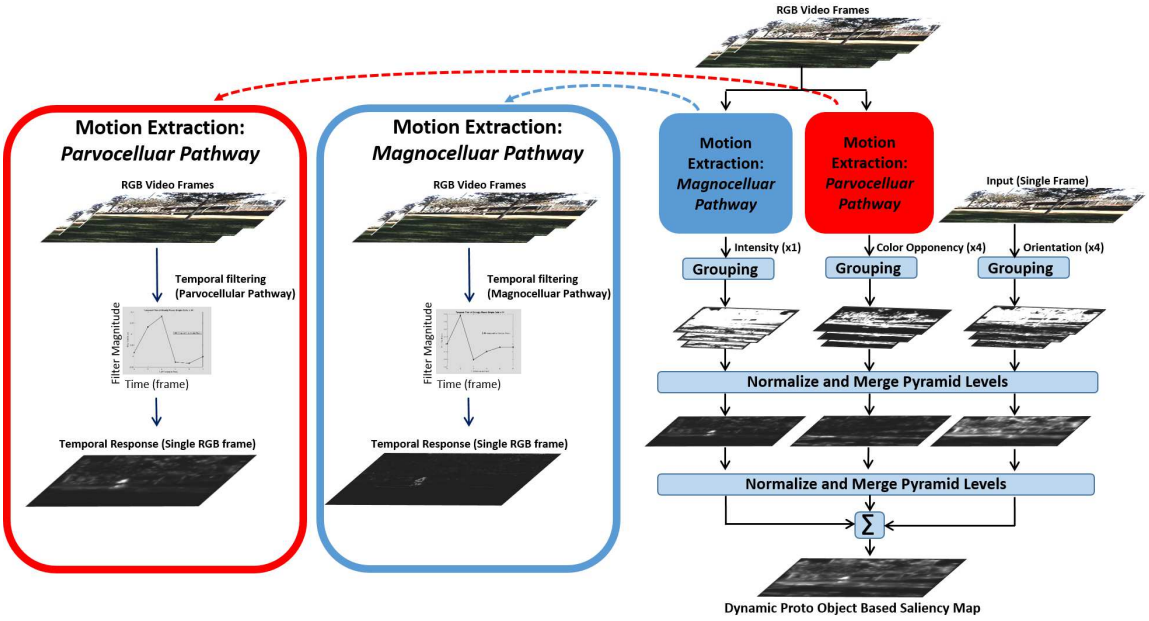


Figure 2.9: DPOVS-ST Model. Dynamic Proto-Object based Visual Saliency - SpatioTemporal filters. This model utilizes spatial and temporal information of the dynamic scene as input to the model. The model receives RGB/color video frames as input. For the intensity channel, motion is extracted using strongly phasic temporal filters (magnocelluar pathway). The RGB output of this filter is the input to the grouping stage. In the color channels, motion is extracted using weakly phasic temporal filters (parvocelluar pathway). The output of this filter is the input to the grouping stage in the color channels. No motion is extracted within the orientation channel so that static information is preserved. Details of this model can be found in Section 2.5.

2.5.1 Feature Channel Extraction

The model receives dynamic visual stimuli (i.e. video) as input. This input can also be seen as a sequence of images (video frames). For each frame, a new saliency map is computed. We assume a resolution of 480 $\times$ 640 and video speed of 24 frames per second.

2.5.1.1 Intensity Channel - Separable Space-Time Filtering

Considering that simple cells in the magnocelluar pathway have low contrast selectivity [63], we apply the strongly phasic temporal filter in the intensity channel. To extract the intensity of the current frame, the average of the red (r), green (g), and blue (b) channel is computed (See Equation 2.2).

$$I = \frac{r + g + b}{3} \tag{2.2}$$

The temporal filter is applied on the current frame and five previous frames of the intensity version (I) of these frames. The convolution is applied temporally across the video frames. The total number of frames in which the convolution is applied is dependent on the frame rate of the videos. In our case, a framerate of 24Hz suffices, which results in a filter convolution over the current frame and 5 previous frames. The representation of this discrete convolution can be seen in Equation 2.3.

$$M[n] = (F * R)[n] = \sum_{t=0}^{T} \sum_{r=1}^{N_r} \sum_{c=1}^{N_c} F_{r,c}[n-t] \times R[t] \tag{2.3}$$

$M[n]$ is the temporal output at frame n. $F_{r,c}[n]$ is the pixel intensity of the original gray-scaled (intensity version) video at row r and column c at frame $F[n]$. $R[t]$ is the discretized filter of $r(t)$ in Equation 2.1 using the strongly (or weakly) phasic parameters in Table 2.1. Finally, $F[n-t]$ represents the frame at t number of frames in the past. T is the total number of frames in the past over which to perform the convolution. In our case, $T = 6$. The output, $M[n]$ is the input to the grouping stage within the intensity channel (strongly phasic output) and color channel (weakly phasic output).

2.5.1.2 Color Channel - Separable Space-Time Filtering

Considering simple cells in the parvocelluar pathway have high contrast selectivity and are less sensitive to motion, the weakly phasic temporal filter is applied within this channel. The same convolution is performed on the video sequence for the red, green, and blue channels. However, in this case, the weakly phasic filter is applied and henceforth, $R[t]$ is modeled by Equation 2.1 using the weakly phasic parameters in Table 2.1. The RGB output after applying this weakly phasic temporal filter is used as input to the color subchannels. The color channel is made up of four subchannels. These are red-green opponency (RG), green-red opponency (GR, blue-yellow opponency (BY), and yellow-blue opponency (YB). These features are extracted by

decoupling hue from intensity by normalizing each color channel by intensity. These subchannels are computed using the temporal output as follows:

$$R = \lfloor r - \frac{g+b}{2} \rfloor \tag{2.4}$$

$$G = \lfloor g - \frac{r+b}{2} \rfloor \tag{2.5}$$

$$B = \lfloor b - \frac{r+g}{2} \rfloor \tag{2.6}$$

$$Y = \lfloor \frac{r+g}{2} - \frac{|r-g|}{2} - b \rfloor \tag{2.7}$$

Using Equations 2.4 to 2.7, the four color opponencies are computed as follows:

$$RG = \lfloor R - G \rfloor \tag{2.8}$$

$$GR = \lfloor G - R \rfloor \tag{2.9}$$

$$BY = \lfloor B - Y \rfloor \tag{2.10}$$

$$YB = \lfloor Y - B \rfloor \tag{2.11}$$

These four separable spatiotemporal filtered outputs (RG,GR,BY,YB) are used as input to the grouping stage of the color channel.

2.5.1.3 Orientation Channel - Spatial Filtering Only

Within the orientation channel, there is no temporal filtering. Extraction of temporal information within the intensity channel and color channel is sufficient. Furthermore, this helps to perserve static information with regards to saliency. In the

orientation channel, there are four subchannels. Within each channel, saliency is computed in regards to salient objects with respect to a unique orientation. These four subchannels are O_0, $O_{\frac{\pi}{4}}$, $O_{\frac{\pi}{2}}$, and $O_{\frac{3\pi}{4}}$ where 0, $\frac{\pi}{4}$, $\frac{\pi}{2}$, and $\frac{3\pi}{4}$ correspond to the four unique orientations. For each of these subchannels, the gray-scaled, intensity version of the current frame (See Equation 2.2) is the input to the grouping stage.

2.5.2 Grouping Stage

The computation performed in the grouping stage was introduced in Section 2.2.3.1. This stage computes proto-objects within each channel, independently, using a biologically-plausible grouping mechanism inspired by Craft et al. [20]. The computational model for the grouping mechanism used in this model can be seen in Fig. 2.10. This is the same model used in Russell et al. [1].

Within each channel, the grouping mechanism takes in input from the feature extraction (post spatial or spatiotemporal filtering). The first step of grouping computation is the edge extraction using 2D Gabor filters [107]. This is representative of the receptive field of simple cells in V1. Even and odd edge filter kernels ($g_{e,\theta}(x,y)$ and $g_{o,\theta}(x,y)$)are used and can be seen in Equations 2.12 and 2.13.

$$g_{e,\theta}(x,y) = e^{-\frac{x'^2+\gamma^2 y'^2}{2\sigma^2}} cos(\omega x') \tag{2.12}$$

$$g_{o,\theta}(x,y) = e^{-\frac{x'^2+\gamma^2 y'^2}{2\sigma^2}} sin(\omega x') \tag{2.13}$$

$$x' = xcos(\theta) + ysin(\theta) \tag{2.14}$$

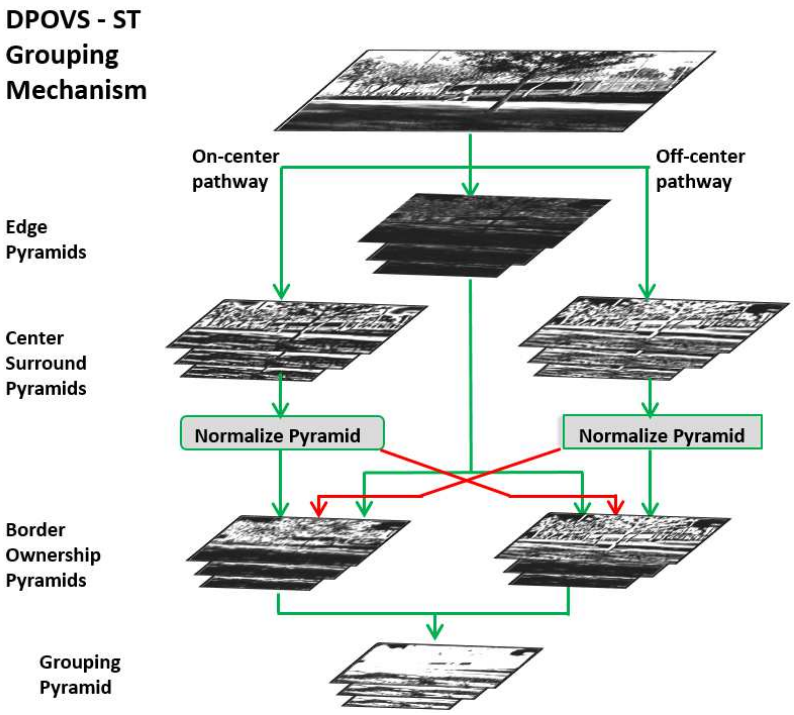


Figure 2.10: Computational grouping mechanism used in DPOVS-ST Model. See Section 2.5.2 for details.

$$y' = -xsin(\theta) + ycos(\theta) \tag{2.15}$$

The value θ has four different values, 0, $\frac{\pi}{4}$, $\frac{\pi}{2}$, and $\frac{3\pi}{4}$ radians. Each of these orientations represent a receptive field sensitivity to a specific oriented edge. The parameter γ is the spatial aspect ratio, σ is the standard deviation, and ω is the spatial frequency. The coordinates (x, y) represent the location of a pixel in the image/frame. The variables x' and y' are the rotated coordinates. The response to these edge filters are noted by Equation 2.16 and 2.17.

$$S^k_{e,\theta}(x, y) = I^k(x, y) * g_{e,\theta}(x, y) \tag{2.16}$$

$$S^k_{o,\theta}(x, y) = I_k(x, y) * g_{o,\theta}(x, y) \tag{2.17}$$

The input image/frame pyramid is $I^k(x, y)$ where k represents a single level in the pyramid. This correlation $(*)$ is applied to each level of the pyramid independently (scale invariance). The correlation operation can be seen in Equation 2.18.

$$f(x, y) * g(x, y) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m, n)g(x + m, y + n) \tag{2.18}$$

In order to achieve contrast invariance from simple cell responses to oriented edges, the complex cell response is computed [108] as seen in Equation 2.19.

$$C^k_\theta(x, y) = \sqrt{S^k_{e,\theta}(x, y)^2 + S^k_{o,\theta}(x, y)^2} \tag{2.19}$$

The next stage is computation of center-surround responses (both ON- and OFF-center). This is necessary for inferring the knowledge of objects in the scene, and later whether or not they belong to figure or background. Within the intensity and

color channels, the ON-center filter kernel used can be seen in Equation 2.20 and the OFF-center filter kernel used can be seen in Equation 2.21.

$$CS_{on}(x,y) = \frac{1}{2\pi\sigma_i^2} e^{-\frac{x^2+y^2}{2\sigma_i^2}} - \frac{1}{2\pi\sigma_o^2} e^{-\frac{x^2+y^2}{2\sigma_o^2}} \tag{2.20}$$

$$CS_{off}(x,y) = -\frac{1}{2\pi\sigma_i^2} e^{-\frac{x^2+y^2}{2\sigma_i^2}} + \frac{1}{2\pi\sigma_o^2} e^{-\frac{x^2+y^2}{2\sigma_o^2}} \tag{2.21}$$

The parameters σ_i and σ_o are the standard deviation of the inner (center) Gaussian and outer (surround) Gaussian, respectively. For computing the center-surround responses in the orientation channel, the Gabor filters are applied with orientations corresponding to the orientation of its subchannel. The equations for the ON- and OFF-center-surround kernels used in the orientation channels can be seen in Equations 2.22 and 2.23.

$$CS_{on}(x,y) = e^{-\frac{x'^2+\gamma_1^2 y'^2}{2\sigma_1^2}} cos(\omega_1 x') \tag{2.22}$$

$$CS_{off}(x,y) = -e^{-\frac{x'^2+\gamma_1^2 y'^2}{2\sigma_1^2}} cos(\omega_1 x') \tag{2.23}$$

Using these center-surround kernels, the center-surround responses are computed using the correlation operator as seen in Equations 2.24 and 2.25 for ON-center and OFF-center responses.

$$CS_L^k(x,y) = \lfloor I^k(x,y) * CS_{on}(x,y) \rfloor \tag{2.24}$$

$$CS_D^k(x,y) = \lfloor I^k(x,y) * CS_{off}(x,y) \rfloor \tag{2.25}$$

The parameter k represents the level in the pyramid, while $CS_D^k(x,y)$ and $CS_L^k(x,y)$ are the responses for dark objects on light background and light objects on dark background, respectively. The operator $\lfloor . \rfloor$ is a half-wave rectification.

The next step involves computing antagonist pairs of border ownership activity (B_θ and $B_{\theta+\pi}$ for a given angle, θ. These are computed by modulating the complex cell response (Equation 2.19) with center-surround responses (Equations 2.24 and 2.25). The border ownership activity is computed independently for light objects and dark objects using Equations 2.26 and 2.27, respectively.

$$B^k_{\theta,L}(x,y) = \left\lfloor C^k_\theta(x,y) \times \left(1 + \sum_{j \geq k} \frac{1}{2^j} v_{\theta+\pi}(x,y) * CS^j_L(x,y) - w_{opp} \sum_{j \geq k} \frac{1}{2^j} v_\theta(x,y) * CS^j_D(x,y)\right)\right\rfloor \tag{2.26}$$

$$B^k_{\theta,D}(x,y) = \left\lfloor C^k_\theta(x,y) \times \left(1 + \sum_{j \geq k} \frac{1}{2^j} v_{\theta+\pi}(x,y) * CS^j_D(x,y) - w_{opp} \sum_{j \geq k} \frac{1}{2^j} v_\theta(x,y) * CS^j_L(x,y)\right)\right\rfloor \tag{2.27}$$

The parameter w_{opp} is the synaptic weight of the inhibitory signal from the opposite polarity CS response. The term 2^{-j} is a factor applied such that influence across spatial scales is constant. The v_θ term is a kernel generated using the von Mises distribution seen in Equation 2.28. It is used for mapping the center-surround response back to its corresponding complex edge response.

$$\begin{aligned} v_\theta(x,y) &= -\frac{e^{\sqrt{x^2+y^2}-R_0} sin(tan^{-1}(\frac{y}{x} - \theta)}{2\pi I_0(\sqrt{x^2+y^2} - R_0)} \\ v_\theta(x,y) &= \frac{v_\theta(x,y)}{max(v_\theta(x,y))} \end{aligned} \tag{2.28}$$

The parameter R_0 is the zero crossing radius of the center-surround kernels and θ is the orientation of the kernel. The parameter I_0 is the modified Bessel function of the first kind. Finally, v_θ is normalized according to its max value. The border ownership

responses for light and dark objects (Equations 2.26 and 2.27) are summed to achieve invariance to the center-surround polarity (Equation 2.29.

$$B^k_\theta(x,y) = B^k_{\theta,L}(x,y) + B^k_{\theta,D}(x,y) \tag{2.29}$$

The contrast invariant border ownership response is computed for both $B^k_\theta(x,y)$ and $B^k_{\theta+\pi}(x,y)$ using Equation 2.29. The next step involves computing the difference, $B^k_\theta(x,y) - B^k_{\theta+\pi}(x,y)$. The magnitude of this difference is representative of the confidence measure of border ownership at pixel (x,y) and the sign of the difference reveals the direction of border ownership at the pixel (x,y). This border ownership computation is supported by the neurophysiological findings in [19,96]. At each pixel, there exist border ownership responses of four different orientations. However, a pixel can only belong to a single border, and henceforth, the winning border ownership response $(\hat{B}^k(x,y))$ is selected as seen in Equations 2.30 and 2.31.

$$\hat{B}^k(x,y) = B^k_{\hat{\theta}}(x,y) \tag{2.30}$$

, where

$$\hat{\theta} = argmax_\theta\big(\big|B^k_\theta(x,y) - B^k_{\theta+\pi}(x,y)\big|\big) \tag{2.31}$$

The final step of the grouping computation is to integrate winning border ownership responses in a circular fashion. This biases the grouping cells (G) to show preference for objects which exhibit continuity and proximity (Gestalt principles).

The equation for computing grouping activity can be seen in Eqaution 2.32.

$$G^k(x,y) = \sum_{\theta} \Big\lfloor \delta\Big(B^k_\theta(x,y), \hat{B}^k(x,y)\Big) \times \Big[B^k_\theta(x,y) - w_b \times B^k_{\theta+\pi}(x,y)\Big] * v_\theta(x,y) \Big\rfloor \quad (2.32)$$

This grouping activity is computed within each of the nine feature channels and each level (k) of each image pyramid, independently. The grouping activity provides notion of proto-objects which exist within the visual stimuli. Furthermore, it provides perceptual organization of the visual scene into figure-ground. This input is fed into the next normalization stage of the model.

2.5.3 Grouping Normalization

A normalization operator (similar to Itti et al. [18]) is applied during the grouping stage to enhance maps with single proto-objects and suppress maps with multiple proto-objects. This normalization operator ($N_1(.)$) is applied to a single level in the pyramid and works as follows:

1. Find the max value, M, across both maps (CS_L and CS_D).

2. Normalize CS_L and CS_D maps simultaneously to a range [0,...,M].

3. Find the average local max, $\bar{m}$ across both maps.

4. $N_1(.) \rightarrow$ Multiply both maps by $(M - \bar{m})^2$.

2.5.4 Final Normalization and Summation

The final steps of the model for computing saliency on the current frame involves first collapsing the grouping activity pyramid within each channel into a single conspicuity map. This is achieve by first performing a similar normalization operator as $N_1(.)$ to each level in the pyramid and then rescaling each level to a common scale and summing across the pyramid. This second normalization operator ($N_2(.)$) is identical to $N_1(.)$ except that it is only applied to a single map. The normalization and collapsing of the grouping pyramid is applied to each individual subchannel. It is then summed across subchannels within each of the three feature channels: $\hat{I}$ (intensity), $\hat{C}$ (color), and $\hat{O}$ (orientation). See Equations 2.33, 2.34, and 2.35.

$$\hat{I} = \sum_{k=1}^{10} N_2(G_I^k) \tag{2.33}$$

$$\hat{C} = \sum_{k=1}^{10} \Big(N_2(G_{RG}^k) + N_2(G_{GR}^k) + N_2(G_{BY}^k) + N_2(G_{YB}^k) \Big) \tag{2.34}$$

$$\hat{O} = \sum_{k=1}^{10} \Big(N_2(G_{O_0}^k) + N_2(G_{O_{\frac{\pi}{4}}}^k) + N_2(G_{O_{\frac{\pi}{2}}}^k) + N_2(G_{O_{\frac{3\pi}{4}}}^k) \Big) \tag{2.35}$$

The final step involves performing the same normalization operator $N_2(.)$ on each conspicuity map and linearly summing these results to form the final saliency map output at the current frame (See Equation 2.36).

$$S = N_2(\hat{I}) + N_2(\hat{C}) + N_2(\hat{O}) \tag{2.36}$$

2.6 Model 2 - DPOVS-BOM (Border Ownership Modulation)

The second model for integrating motion into this proto-object based visual saliency model is based on the notion that border ownership activity is modulated by motion. It is called Dynamic Proto-Object based Visual Saliency - Border Ownership Modulation (DPOVS-BOM). It also computes a dynamic visual saliency map as a function of proto-objects and their temporal information. This idea supports the idea that features moving together should be grouped together. Henceforth, regions exhibiting motion will have a greater notion of object/figure. This enhances saliency with respect to proto-object activity when the proto-objects are exhibiting motion. Similarly, to the previous model (DPOVS-ST), motion is extracted using biologically-plausible strongly phasic and weakly phasic temporal filters. Within the intensity channel, a strongly phasic temporal filter is applied to extract motion. This motion information is then used to multiplicatively modulate border ownership activity in its corresponding channel. In the same manner, the weakly phasic temporal filter output is used to modulate border ownership activity within the color channel. The orientation is not modulated by motion and the intensity version of the current frame is input to the orientation channel. The model can be seen in Fig. 2.11.

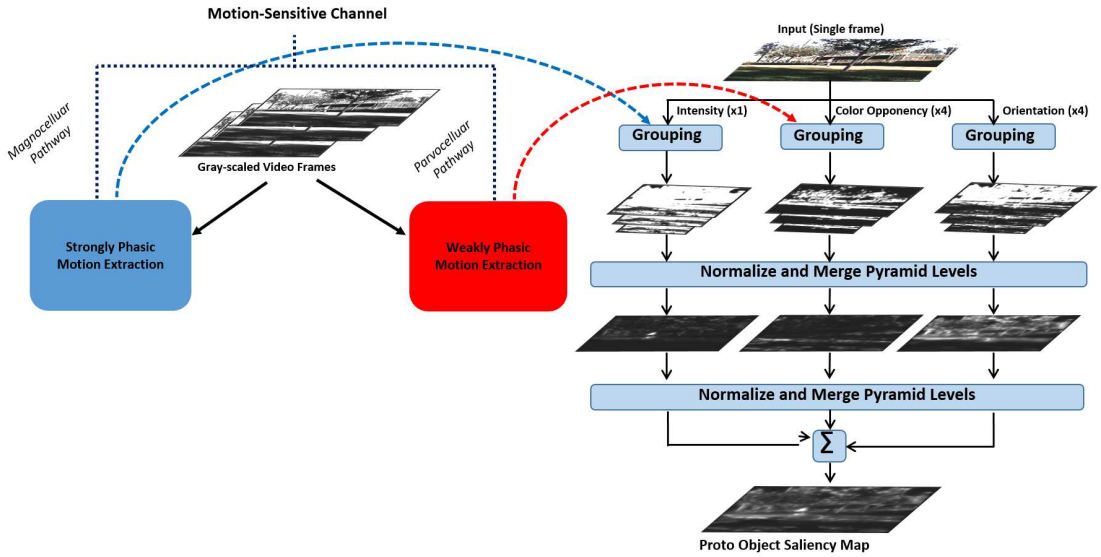


Figure 2.11: DPOVS-ST Model. Dynamic Proto-Object based Visual Saliency - Border Ownership Modulation. This model utilizes spatial and temporal information of the dynamic scene as input to the model. The model receives RGB/color video frames as input. For the intensity channel, motion is extracted using strongly phasic temporal filters (magnocelluar pathway). In the color channels, motion is extracted using weakly phasic temporal filters (parvocelluar pathway). The output of these filters is used to modulate border ownership activity within each corresponding channel. Details of this model can be found in Section 2.6.

2.6.1 Feature Channel Extraction

The feature extraction within each channel only occurs in the spatial domain at the current frame. Rather than using the motion output as direct input to the grouping stage, motion is computed in an independent motion-sensitive channel and modulates the border ownership activity. As in the previous model, there are three main feature channels in which proto-objects and saliency are computed (intensity, color, and orientation). Intensity has one subchannel, color has four subchannels (red-green, green-red, blue-yellow, and yellow-blue opponency), and orientation has four subchannels (orientations: 0, $\frac{\pi}{4}$, $\frac{\pi}{2}$, and $\frac{3\pi}{4}$). The Equations/methods for extraction each of these spatial features can be found in Section 2.5.1.

2.6.2 Motion-Sensitive Channel

The motion-sensitive channel in this model is responsible for extracting motion. It extracts motion along two pathways, a magnocelluar pathway and parvocelluar pathway. Along the magnocelluar pathway, a strongly phasic temporal filter is used identical to that seen in the previous model. The strongly phasic motion output is then used to modulate the border ownership activity in the intensity channel. Along the parvocelluar pathway, a weakly phasic temporal filter is used identical to that seen in the previous model. The weakly phasic motion output is then used to modulate the border ownership activity. The temporal filter can be seen in Equation

2.1 and the application of this filter for extraction motion can be seen in Equation 2.3. The integration of the motion-sensitive channel can be seen in Fig. 2.11 and how it modulates border ownership activity can be seen in Fig. 2.12.

2.6.3 Modified Grouping Computation

The grouping computation is similar to that in the previous model. The idea of border ownership neurons communicating with grouping cells still persists. However, the computation in border ownership activity is modified. In this model, there is an additional step involving multiplicative modulation of border ownership activity prior to grouping. Otherwise, the mathematics involved in the grouping computation is identical to that in Section 2.5.2. The complete grouping mechanism used in this model can be seen in Fig. 2.12.

The motion output is computed as seen in Equation 2.37

$$M[n] = (F * R)[n] = \sum_{t=0}^{T} \sum_{r=1}^{N_r} \sum_{c=1}^{N_c} F_{r,c}[n-t] \times R[t] \tag{2.37}$$

$M[n]$ is the temporal output at frame n. $F_{r,c}[n]$ is the pixel intensity of the original gray-scaled (intensity version) video at row r and column c at frame $F[n]$. $R[t]$ is the discretized filter of $r(t)$ in Equation 2.1 using the strongly (or weakly) phasic parameters in Table 2.1. Finally, $F[n-t]$ represents the frame at t number of frames in the past. T is the total number of frames in the past over which to perform the

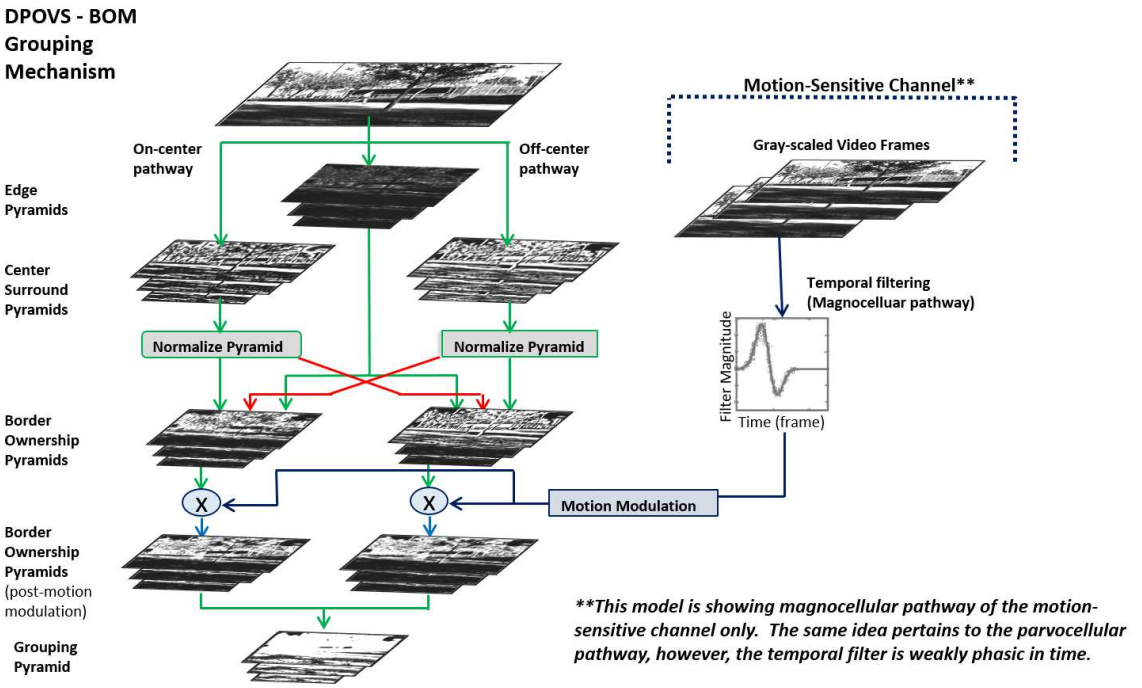


Figure 2.12: Computational grouping mechanism used in DPOVS-BOM Model. This shows only the magnocelluar pathway of the motion-sensitive channel. The parvocelluar pathway is the same, however, the weakly phasic temporal filter is used instead. See Section 2.6.3 for details.

convolution. In our case, $T = 6$. The output, $M[n]$ is the input to the grouping stage within the intensity channel (strongly phasic output) and color channel (weakly phasic output).

Within the intensity and color channels, the motion output is first normalized to a range between 0 and 1 using Equation 2.38.

$$M(x,y) = \frac{M(x,y)}{max\big(M(x,y)\big)} \tag{2.38}$$

In Equation 2.38, we assume $M(x,y)$ is the motion output at the current frame ($M[n]$ from Equation 2.37). This motion is computed at each pyramid level forming the motion output pyramid, $M^k(x,y)$. After normalization, the motion output ($M^k(x,y)$) is used to modulate border ownership activity, $\hat{B}^k(x,y)$ as seen in Equation 2.39.

$$\hat{B}^k_{mod}(x,y) = \hat{B}^k(x,y) \times (1 + w_{motion} \times M^k(x,y)) \tag{2.39}$$

Equation 2.39 is used within both the intensity and color channels. However, within the intensity channel, the motion output, $M^k(x,y)$, is that from the strongly phasic filter. Within the color channel, the motion output, $M^k(x,y)$, is that from the weakly phasic filter. This modulated border ownership activity ($\hat{B}^k_{mod}(x,y)$) is then used for computing the grouping activity as seen in Equation 2.32.

2.6.4 Final Saliency Map Computation

After grouping activity is computed within each channel, the same normalization operations are applied. The normalization operator, $N_1(.)$ is applied to the grouping activity and the $N_2(.)$ normalization operator is applied to the conspicuity maps within each individual channel and to form the final saliency map as in the previous model.

2.7 Validating the Models

To validate the models' accuracy in computing dynamic visual saliency maps, we quantify their capability of predicting human eye fixations over time. This is performed by computing the dynamic saliency map using the model and measuring how well it predicts human eye fixations over time. The metrics used for quantifying the saliency maps ability to predict human eye fixations is the Kullback-Leibler Divergence (KLD).

2.7.1 Kullback-Leibler Divergence (KLD) Metric

The Kullback-Leibler Divergence is a metric used for quantifying the extent one distribution deviates from another. The KLD metric we used is that used in [21]. Each video has corresponding eye fixation/saccade data for a given subject. For each saccade, three different samples are taken. The first sample, S_{sac}, is the max

value within a 3×3 neighborhood (corresponding to an aperture of 5.6° around the location of the subjects target saccade at the time the saccade begins. The second sample, S_{rand}, is the max value within a 3×3 neighborhood around a randomly selected location (from a uniform distribution) at the same frame. Therefore, a high value of S_{sac} represents a high saliency at the target location of the current saccade. Sampling S_{rand} serves as a control for validating saccades. Finally, S_{max} is sampled, which is the max saliency value at the frame of when the saccade begins. From these samples we obtain two distributions. The first distribution is obtained by grouping the ratio of samples S_{sac}/S_{max}. Ideally, a highly-predictive model will have a value of 1.0 for each sample ratio. The second distribution is obtained by grouping the ratio of samples S_{rand}/S_{max}. We then use these two distributions to compute the KL divergence between the two distributions. This is computed by essentially computing the relative entropy of the histogrammed distribution S_{sac}/S_{max} with respect to the histogrammed distribution S_{rand}/S_{max} (See Equation 2.40. A KL divergence equal to 0 suggests there is no difference between saliency at human saccade targets and that of random saccade targets. A higher KL divergence value means the model was further from chance, and henceforth better, at predicting human saccades. We obtain 100 random saccades for each sample in order to obtain 100 different KL divergence values. This allows us to obtain an accurate and reliable KL divergence value.

$$KLD(P,Q) = \sum_i P(i) \ln\left(\frac{P(i)}{Q(i)}\right) \tag{2.40}$$

$$P(i) = \frac{S_{sac}(i)}{S_{max}(i)} \tag{2.41}$$

$$Q(i) = \frac{S_{rand}(i)}{S_{max}(i)} \tag{2.42}$$

The values $P(i)$ and $Q(i)$ are probability distributions as denoted by Equations 2.41 and 2.42. The variable i corresponds to a saccaade/fixation.

2.7.2 Dataset and Comparison

To validate this model we used the dataset, "CRCNS Data Sharing: Human eye-movements under natural free viewing", from Itti et al. [21] from University of Southern California. In these experiments, 8 subjects (5 male, 3 female age 23-32) eye fixation data was recorded for 50 video clips. These videos were natural dynamic scenes ranging from TV commercials to video games. The videos were displayed at a resolution of 640×480 with a frame rate of 30.1341 Hz. The eye tracker sampled at a rate of 240 samples/sec. The display used was a 22-inch CRT monitor at 80cm viewing distance.

In our first experiments we compared the DPOVS-BOM model to other state-of-the-art. These models were the dynamic version of the Graph-based Visual Saliency model (GBVS, 2006) [80] and the Itti et al. model (Itti, 2005) [21]. Each of these models have a motion component and were sufficient for comparison with the DPOVS-BOM model. The first experiments involved using 4 arbitrary videos from this dataset and 4 subjects' eye fixation data corresponding to these videos. The KL divergence was computed for our model (DPOVS-BOM), GBVS (2006) model, and Itti (2005) model and compared. The significance of these results was calculated using a paired t-test between the distribution of KLD values for each repetition of KLD computation. The resulting p-value suggested the rejection of the null hypothesis that these two sets could have been drawn from distributions with the same mean. A p-value ¡ 0.05 represents a significant result.

The second experiment involved comparing our DPOVS-ST model with the DPOVS-BOM model. For this experiment, 5 arbitrary videos were used with 4 subjects' eye fixation/saccade data for each video. Similarly to the first experiment, we computed the KL divergence from random saccades for each model on each video independently. The p-value was also computed for quantifying the significance of the results.

2.8 Results and Discussion

The results form the first experiment can be seen in Table 2.2. Using each models computed dynamic saliency map, for each video, each models KLD was measured for each subject using the method previously described. Table 2.2 shows the results of the KLD measurements (scores). These results reflect the average KLD over all subjects eye fixation data and further averaged over all 4 videos. These subjects were focused on the center of the screen prior to starting the video, causing the eye fixation data to have strong center bias at the initiation of the video. Top-down factors are also not considered in this model which also effect the subjects eye movements. Henceforth, these KLD values may underestimate the models true performance for predicting bottom-up attention. Nonetheless, this proto- object based model outperforms the Itti (2005) model (+0.1009) and insignificantly differs from the GBVS (2006) model (+0.0165) in performance with respect to the computed KLD in this experiment. Moreover, this model performs significantly better than chance (chance being a KLD score of 0) in predicting eye movements. Significance was calculated using a paired t-test between the results of the proto-object model and the Itti (2005) model, as well as between the proto-object model and the GBVS (2006) model. The p-values seen in Table 2.2 indicate that the results between the proto-object model and Itti (2005) model are highly significant while the results between the GBVS (2006) model are much less significant. Such p-values are expected considering only 4 videos were used from the dataset. This work and results were published in [83].

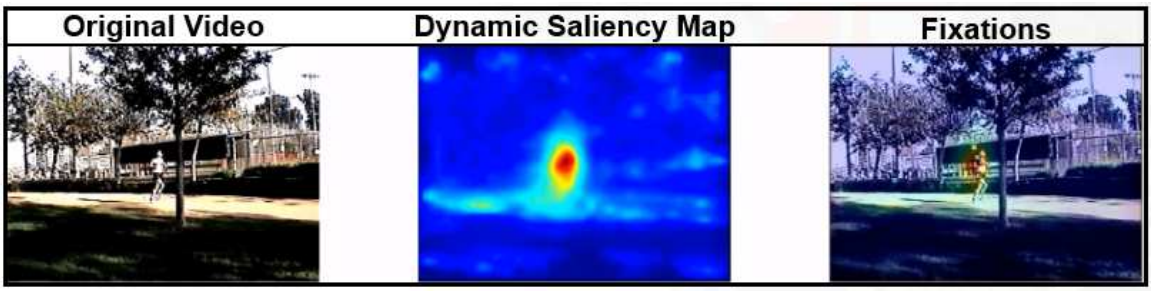


Figure 2.13: Original video is on the far left, the dynamic saliency map output from our model is in the middle, and the far right shows fixation data overlaid on the original video as well as the saliency map overlaid (Video: 'beverly08' from Itti (2005) dataset [21]).

Table 2.2: Average KL Divergence and Significance of DPOVS-BOM in Comparison to Other SOTA Models

Metric	**DPOVS-BOM**	**Itti (2005)**	**GBVS (2006)**
KLD	0.41902	0.31811	0.40257
p-value	-	0.0676	0.7884

The results from the second experiment can be seen in Table 2.3. From these results, we can conclude that the Model 1 (DPOVS-ST), which utilizes space-time separable spatiotemporal filters, performs better than the Model 2 (DPOVS-BOM), which uses temporal information to modulate border-ownership activity. Both models predicted human saccades with a KLD better than chance (chance being a KLD = 0) for each video. However, the DPOVS-ST model achieved a higher average KLD value for all videos except for the video labeled tvads04. Considering the p-values for each video, our results are significant for all videos except for beverly08 and tvads04. The reason the p-values were not lower was due to the fact that we selected 5 videos with 4 subjects saccade data per video. However, the results were sufficient enough to conclude that DPOVS-ST, which uses space-time spatiotemporal filters, was better at predicting saccades than that of the DPOVS-BOM model. It is clear that the KLD values for the DPOVS-BOM were higher for all the videos except for tvads04. However, the p-value for that video was 0.1556, which means the results were not highly significant. The results for video beverly08 suggest a higher KLD value for the DPOVS-ST model. However, the p-value for this video was not highly significant (p-value = 0.5103). Overall, these are promising results depicting that the DPOVS-ST, based on the idea of separable spatiotemporal filters, is better at predicting human saccades than that of the DPOVS-BOM, based on border-ownership modulation. This work and results were published in [84].

Table 2.3: Average KL divergence and corresponding p-value for validating each model's ability to predict human saccades.

Video Sequence	**KLD-Model 2: DPOVS-BOM**	**KLD-Model 1: DPOVS-ST**	***p*-value (Significant?)**
'beverly08'	0.1599	0.1653	0.5103 ✗
'monica06'	0.2013	0.2375	0.0072 ✓
'standard01'	0.2723	0.2969	0.0167 ✓
'standard03'	0.1719	0.1927	0.0048 ✓
'tvads04'	0.4180	0.4010	0.1556 ✗

It is important to note that we should not completely rule out the DPOVS-BOM model. Although not as biologically-plausible, the DPOVS-BOM model allows for a tuning weight parameter, w_{motion}. This allows for variability in the effect motion has on the model. Some applications can take advantage of this model when the effect of motion may need to dynamically change. This DPOVS-BOM model still performed better than other SOTA models in predicting human eye saccades.

2.9 Proto-object Based Visual Saliency in Warped Space

2.9.1 Motivation

There has been recent interest in computing saliency in warped space. By warped space, we refer a spatial domains such as 3-dimensional space for virtual reality or raw output of a camera with a panoramic, 360° ("fish-eye") lens. Remote Reality [109] has expressed interest in such a dynamic saliency model which can compute saliency on the "warped" 360° camera output. Such a model requires modification of the current proto-object based saliency model to compensate for the computation in warped space. When performing any visual processing task on these 360° images/frames, there is a large cost in dewarping the camera output 2-D, rectangular coordinate space prior to performing the task (i.e. object recognition, tracking, etc.). Henceforth, the capability to compute saliency on the warped output and then dewarp only interesting, foreground regions of the visual scene would allow for enormous savings in computation cost. Rather than dewarping all of the pixels, only pixels deemed salient will be dewarped and processing will be performed on these interesting regions as it is unnecessary to dewarp and process background, irrelevant information. An example of Remote Reality warped output can be seen in Fig. 2.14.

Figure 2.14: Example of 360° lens camera output from Remote Reality virtual reality device while snowboarding (post-dewarp). Such an output consists of large amount of data and dewarping all of this data at each frame is extremely costly. Henceforth, we seek to compute saliency in this warped space and only dewarp regions deemed salient. This dramatically reduces computation cost.

2.9.2 Input: Warped vs. Dewarped

In this work, we use a Sony Bloggie Camera with a omni-directional, 360° lens. An example of the raw "warped" output of this camera can be seen in Fig. 2.15. This is result of using the "fish-eye" lens. Using the optics of the lens of this camera, the warped output can be "dewarped" to its rectangular coordinate representation as seen in Fig. 2.16. We seek to modify the proto-object based visual saliency model to compute saliency on the warped camera output prior to dewarping.

2.9.3 Modification of Filters

The modification that must be made to the proto-object based visual saliency models is the spatial location of the filter kernels used in the models. The original model is designed for 2-D images on the rectangular coordinate system. In this

Figure 2.15: Sony Bloggie Camera output example (Warped).

Figure 2.16: Sony Bloggie Camera output example (Dewarped).

original model, for each convolution, a single, constant kernel can be used as a sliding window as it is convolved over the image/frame. To perform these convolutions over the warped image, the filter kernel must also be warped at each pixel location. The results in a different kernel to be used at each pixel location. The optics of the lens determines how the kernel is warped at each location. In the proceeding section we discuss the optics of the lens and how we use it to warp the kernel at each pixel location.

2.9.3.1 Bloggie Camera Omni-directional Lens Optics

The raw output of the omni-directional lens of the Bloggie camera is the "fish-eye", warped representation (See Fig. 2.15). The objective is to map from 2-D, rectangular coordinates to polar coordinates. In other words, for any given pixel location in rectangular coordinates (dewarped representation), (x_{rect}, y_{rect}), it should be mapped to its corresponding location in polar coordinates (warped representation), (x_{warp}, y_{warp}). This idea can be seen in Equation 2.43 and visualized in Fig. 2.17.

$$\textit{Mapping: } (x_{rect}, y_{rect}) \rightarrow (x_{warp}, y_{warp}) \tag{2.43}$$

Given the coordinates of the center point of the warped output, $(imgCx, imgCy)$, the x_{warp} and y_{warp} locations can be computed as seen in Equations 2.44 and 2.45.

$$x_{warp} = imgCx + rcos(\theta) \tag{2.44}$$

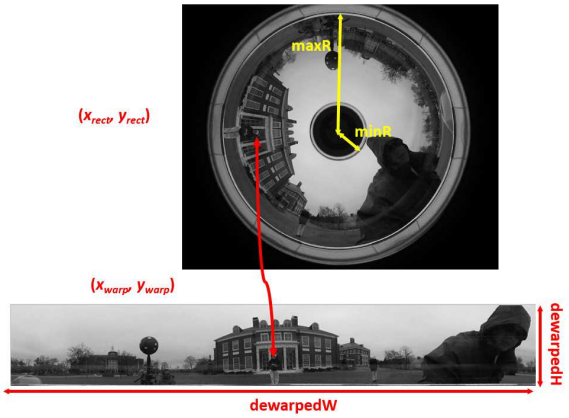


Figure 2.17: Sony Bloggie Camera optics visualization (See Section 2.9.3.1).

$$y_{warp} = imgCy + rsin(\theta) \tag{2.45}$$

The parameter θ is the angle at which the coordinate lies in the warped representation of the image. It is calculated in radians as a function of $dewarpedW$ (width of dewarped representation) and an $angularShift$ offset (See Equation 2.46. The parameter r is the radius. It is a function of the max radius of the warped image ($maxR$) and inner radius ($minR$) and is modulated by the parabolic lens structure (y_{frac}) and can be seen in Equation 2.47. The parabolic function (y_{frac}) for modeling this refraction is a function of y_{rect} and can be seen in Equation 2.48.

$$\theta = \left(0 - \left(\frac{x_{rect}}{dewarpedW}\right)\right) + angularShift \tag{2.46}$$

$$r = \left(y_{frac} \times (maxR - minR)\right) + minR \tag{2.47}$$

$$y_{frac} = yWarpA \times y_{frac}^2 + yWarpB \times y_{frac} + yWarpC \tag{2.48}$$

The parameter values can be seen in Table 2.4.

Table 2.4: Parameter Values for Bloggie Lens

$yWarpA$	$yWarpB$	$yWarpC$	$angularShift$
0.1850	0.8184	-0.0028	0.0

2.9.3.2 Warped Filter Kernels

These equations modeling the camera lens for converting from rectangular coordinates (x_{rect}, y_{rect}) to warped space coordinates (x_{warp}, y_{warp}) are used for warping the filter kernel. When performing a filter/convolution in warped space, at each pixel, when a weighted sum is computed, a unique warped kernel is used. This warped kernel is a function of $maxR$, $minR$, lens refraction parameters (y_{frac}), the current pixel location (in rectangular coordinates) at which the weighted sum is being computed, and finally, the kernel itself. When warping the filter kernel to warped space, there is a reduction in resolution. To overcome this, interpolation is performed on the kernel in dewarped space prior to warping the kernel into warped space. For example, a kernel that was previously 17×17 in size, will undergo nearest-neighbor interpolation and result in 27×27 in size. It is this 27×27 size kernel that is warped into its

kernel representation in warped space and becomes a smaller kernel (e.g. 8×17). An example of such a kernel (Gaussian kernel) can be seen in Fig. 2.18. The 27×27 Gaussian kernel at location (21,61) in dewarped space is now warped to location (506, 1109) using Equations 2.44 and 2.45. The new size of the warped kernel is 8×19.

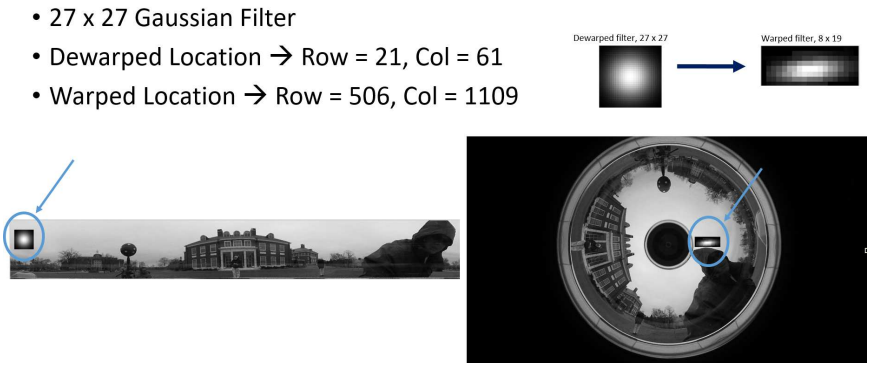


Figure 2.18: Example showing how filter kernel is warped into warped space.

For each filter performed in the proto-object based visual saliency model (e.g. center-surround filtering, edge filtering, etc.), the kernel must be warped in warped space at each pixel location. The convolution then occurs in warped space and the same normalization operators are applied for computing saliency.

2.9.4 Results and Discussion

To verify the kernels are being properly warped, we performed basic smoothing and edge filtering considering the results can be visually confirmed. Results for performing a 7×7 Gaussian smoothing filtering task in warped space can be seen in Fig. 2.19. Results for performing a 3×3 Sobel edge filtering task in warped space can be seen

in Fig. 2.20.

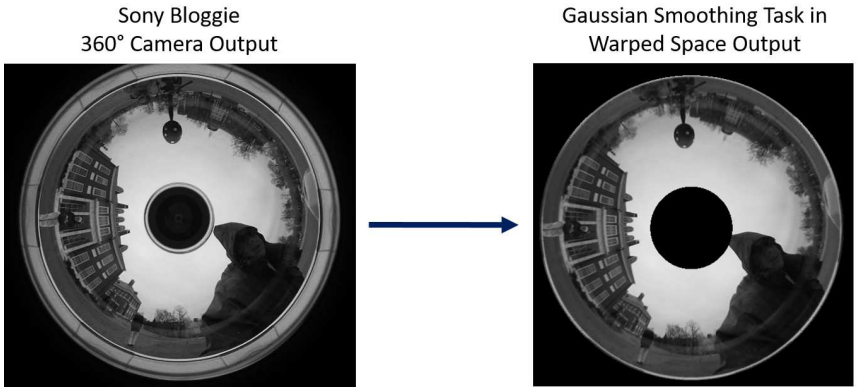


Figure 2.19: Verification of smoothing task in warped space. Gaussian kernel of size 7×7 *convolved with the warped image in warped space by warping the kernel at each pixel location prior to computing the weighted sum.*

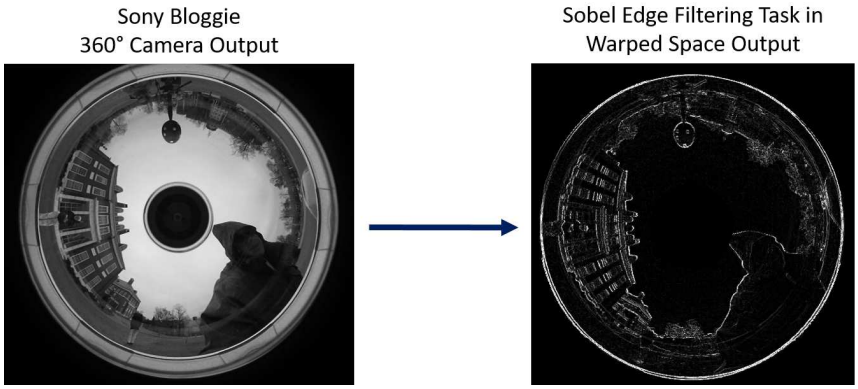


Figure 2.20: Verification of edge detection task in warped space. Sobel edge kernel of size 3×3 *convolved with the warped image in warped space by warping the kernel at each pixel location prior to computing the weighted sum.*

To verify the ability to compute saliency in warped space, we compare results to saliency (using original proto-object based saliency model) computed on the dewarped

version of the image. This is the gold standard to which we compare the output of the modified proto-object based model on the warped version of the image. Each example below uses warped images scaled to 100×100 resolution with a $maxR = 50$ and $minR = 13$, and the dewarped version is of resolution 50×472. From this point on, we will denote the original proto-object based visual saliency model as POVS and the modified proto-object based visual saliency model for warped input as POVS-WARP.

2.9.4.1 Testing (Examples)

The first example warped image used for validating the model can be seen in Fig. 2.21. The dewarped version of the warped image is also shown. It is used for validating our results. A second example can be seen in Fig. 2.22.

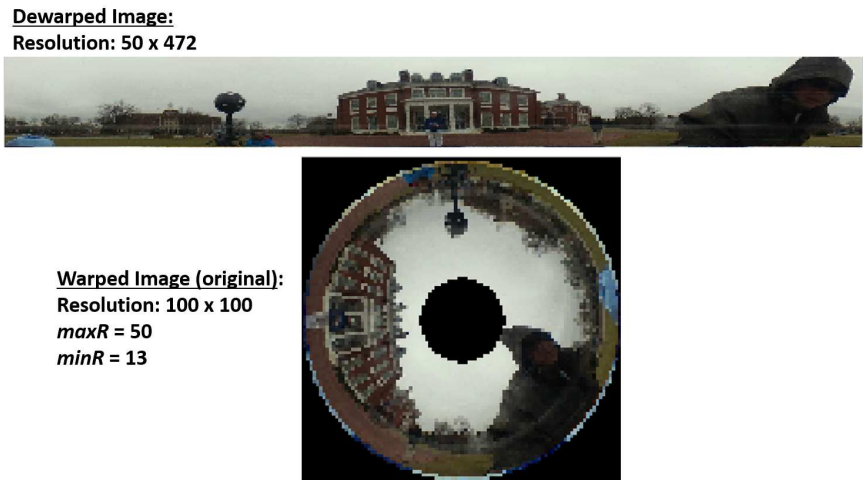


Figure 2.21: Example 1: Warped Input (and Dewarped version)

The first attempt was to run the original model (POVS) on the warped image

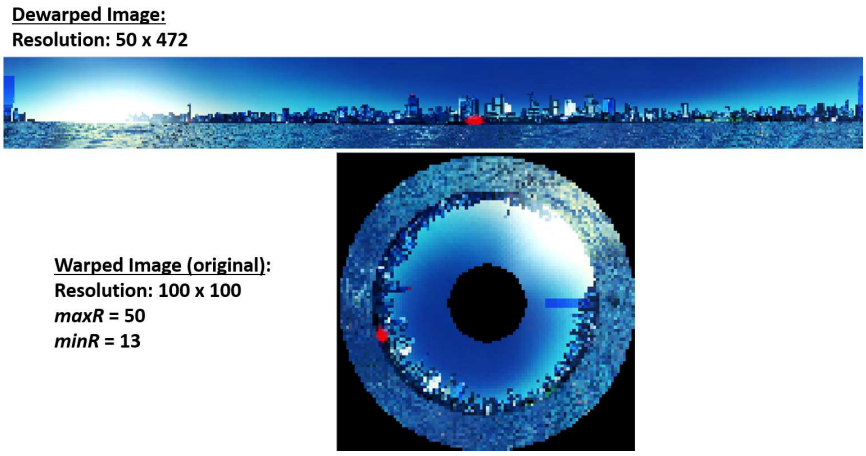


Figure 2.22: Example 2: Warped Input (and Dewarped version)

and then dewarp the resulting saliency map. This result was compared to running the original model (POVS) on the already dewarped image ("gold standard" for validation). These results were insufficient and showed to be unreliable as it does not visually match the "gold standard" (See Fig. 2.23). A second example using these same methods can be seen in Fig. 2.24.

We then ran the modified proto-object based visual saliency for warped images (POVS-WARP) on the original warped image and then dewarped its resulting saliency map. These results were sufficient and visually matched the "gold standard". These results can be seen in Fig. 2.25. A second example using these same methods can be seen in Fig. 2.25

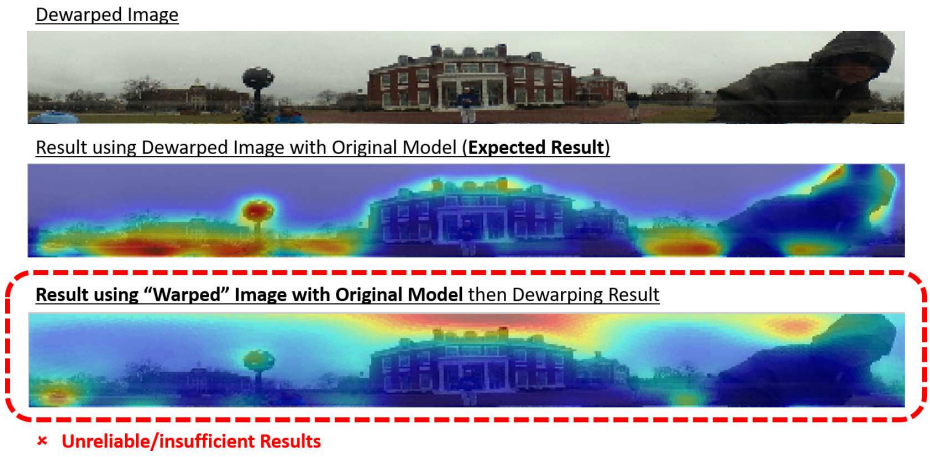


Figure 2.23: Example 1: Comparison of running original POVS model on already dewarped version of image ("gold standard") to that of running the original POVS model on the warped version of the image and then dewarping the result. Results are insufficient.

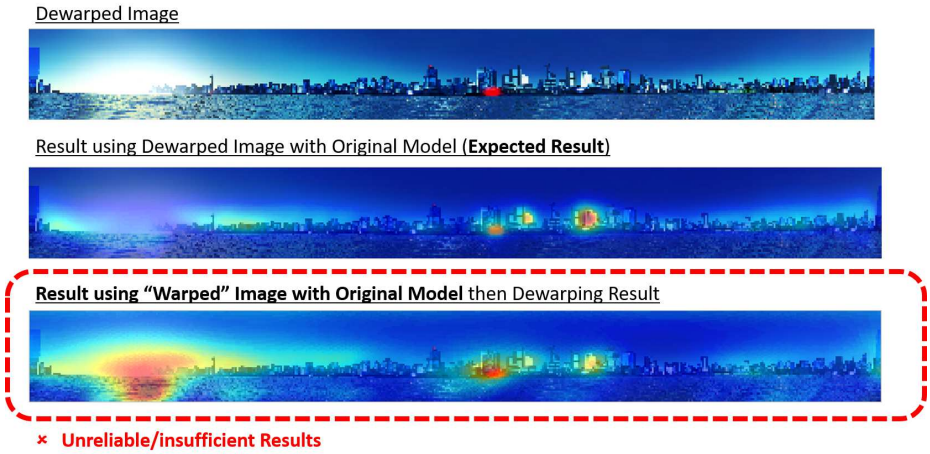


Figure 2.24: Example 2: Comparison of running original POVS model on already dewarped version of image ("gold standard") to that of running the original POVS model on the warped version of the image and then dewarping the result. Results are insufficient.

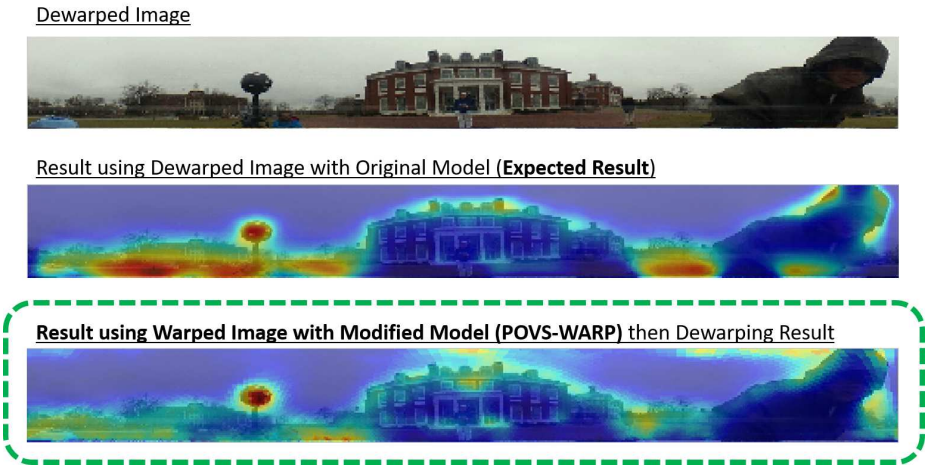


Figure 2.25: Example 1: Comparison of running original POVS model on already dewarped version of image ("gold standard") to that of running the modified model (POVS-WARP) on the warped version of the image and then dewarping the result. Results are sufficient.

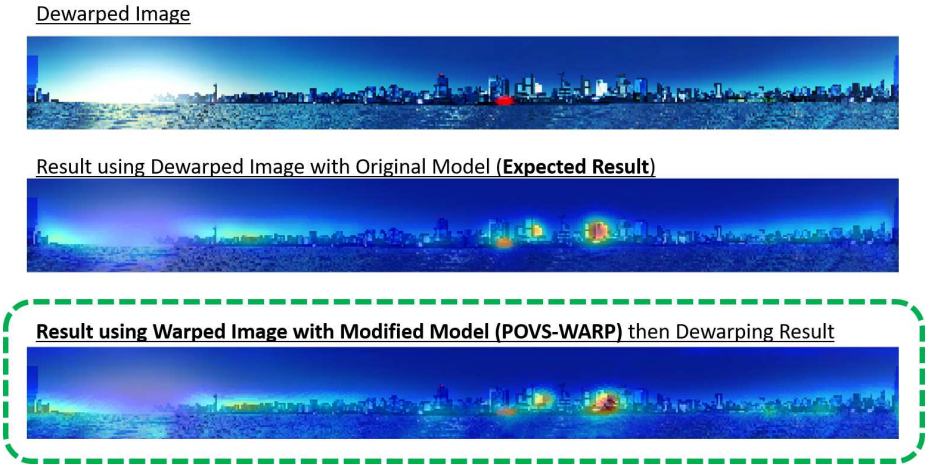


Figure 2.26: Example 2: Comparison of running original POVS model on already dewarped version of image ("gold standard") to that of running the modified model (POVS-WARP) on the warped version of the image and then dewarping the result. Results are sufficient.

2.9.5 Computational Savings

This modified proto-object based visual saliency for warped images (POVS-WARP) poses many advantages. The first is that it does not require dewarping of the complete image/frame prior to processing. Saliency can be computed in the raw warped camera output state and only salient regions can be dewarped. The filter kernel warping only needs to be computed once for each pixel location and then stored in memory. However, this results in a trade-off between this increased speed and decreased computational cost per frame with increased memory requirement (for storing the kernels).

The model parameters can be seen in Table 2.5.

Table 2.5: POVS-WARP Model Parameters

Parameter	**Value**
Resolution	1200×1920
Kernel Size	13×13
# Kernels	14
# Pyramid Levels	8
Kernel Precision	8 `bits`

The additional memory requirement for storing the kernels for the POVS-WARP model can be seen in Equation 2.49.

$$\texttt{Memory Requirement} = Kernel_{precision} \times 13 \times 13 \times \sum_{p=1}^{8}(W_p \times L_p) \tag{2.49}$$

The parameter $Kernel_{precision}$ is the number of bits per kernel coefficient. The parameters W_p and L_p are the width and length of the image in pyramid level p.

The additional computation time for using the original POVS model and needing to dewarp each pixel prior to computation can be seen in Equation 2.50.

$$\texttt{Additional Computation Time (POVS)} = T_{dewarp} \times W_1 \times L_1 \tag{2.50}$$

The parameter T_{dewarp} is the time to dewarp a single pixel. The parameters W_1 and L_1 are the width and length of the top level of the pyramid.

The reduction of computation time can be quantified by the percentage of the warped image deemed salient ($Percent_{sal}$). For example, if only 25% of the current frame/image is deemed salient, the computation time is reduced by 75% considering only 25% of the frame is dewarped opposed to the complete image/frame (See Equation 2.51.

$$\texttt{Additional Computation Time (POVS-WARP)} = Percent_{sal} \times T_{dewarp} \times W_1 \times L_1 \tag{2.51}$$

[1] Assuming 25 percent of image deemed salient.

Table 2.6: Comparison of POVS-WARP vs. POVS (Original) for Saliency Computation on Warped Input

Model Specification	**POVS**	**POVS-WARP**
Kernel Memory Requirement	$\sim 2.37KB$	$\sim 2.37KB \times \sum_{p=1}^{8}(W_p \times L_p)$
Computation Time	$T_{dewarp} \times W_1 \times L_1$	0.25 [1] $\times T_{dewarp} \times W_1 \times L_1$

The memory and computation trade-offs are summarized in Table 2.6. Overall, there is an extreme computation savings when only salient regions are dewarped at each frame of the warped input video sequence opposed to needing to dewarp the entire image at each frame prior to performing processing. The amount of savings is a function of of the percentage of the frame/image deemed salient at the current warped frame ($Percent_{sal}$). In Table 2.6, we assume 25 percent of the image is deemed salient, hence saving 75 percent in computational cost in regards to time and resources.

2.10 Conclusion

In this Chapter, we introduced a novel dynamic proto-object based visual saliency model. Two novel approaches were suggested, POVS-BOM and POVS-ST. Both models outperformed state-of-the-art dynamic visual saliency models in predicting human saccades, and further, the POVS-ST model (based on spatiotemporal filters

as input) outperformed the POVS-BOM model (motion modulates border ownership activity). However, the POVS-BOM model may still be advantageous as it allows for a tuning parameter that allows for increasing or decreasing effect on motion. We further introduce a novel proto-object based visual saliency for warped input which has many interesting applications. We demonstrate a modified model (POVS-WARP) for computing saliency in warped space. Such a model can be applied to warped spaces such as virtual reality or applications utilizing cameras with 360° lens. Computation time is dramatically reduced as only salient regions are dewarped opposed to the entire image prior to processing. The dynamic proto-object visual saliency models POVS-BOM and POVS-ST were published at both Biomedical Circuits and Systems (BioCAS) 2013 in Rotterdam, Netherlands and Conference on Information Systems and Sciences (CISS) 2015 in Baltimore, MD, respectively. In the proceeding chapter (Chapter 3, a novel hardware implementation of a modified dynamic proto-object based visual saliency model on FPGA is discussed.

Chapter 3

FPGA Model: Dynamic Proto-object Based Visual Saliency

3.1 Overview

In the previous chapter, we discussed a novel dynamic proto-object based visual saliency model for computing saliency on video input. In this chapter, we discuss a novel implementation of this model on hardware via FPGA (Field-Programmable Gate Array). We will discuss the details of this digital, hardware implementation on FPGA for real-time processing. FPGA specifications and resources used in implementing the model will also be discussed. We expect that this work may serve as a real-time visual saliency computing system and may inspire future real-time visual saliency model implementations. Furthermore, this work serves as an ideal

pre-processor to systems that perform visual tasks such as object detection and recognition by filtering out background regions of visual stimuli, passing through only the salient regions of an image for further processing. In doing so, this reduces throughput and increases processing speeds.

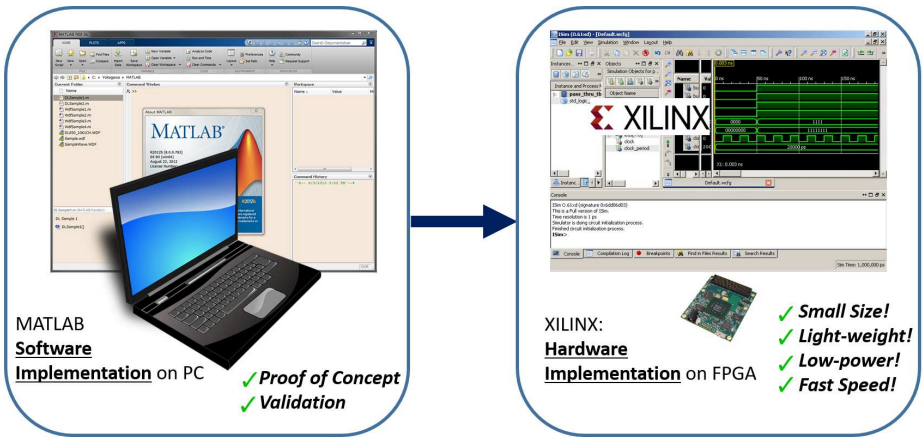


Figure 3.1: The MATLAB implementation discussed in Chapter 2 is ideal for showing proof of concept and validation. Ideally, we want a system implemented in hardware (in this case, FPGA) for small-size, light-weight, low-power, and fast speed applications. We discuss the DPOVS-ST model implemented in hardware in this chapter.

In the remaining of this chapter, we will discuss related work followed by the modifications of the model for FPGA implementation. The FPGA implementation details will then be discussed. Finally, we will show results and compare to the MATLAB implementation with regards to accuracy and speed. This model was demoed at Biomedical Circuits and System in Atlanta, GA in 2015 [22].

3.2 Motivation and Related Work

Considering the computational complexity of this model, we accelerated the computation of the dynamic visual saliency map using a novel FPGA implementation. This allowed for real-time processing of an accurate, biologically-plausible dynamic visual saliency model that is capable of predicting human eye saccades better than other state-of-the-art models. Such real-time processing allows for integration with other visual processing systems that require such fast processing including object recognition and detection.

All of the computational visual saliency models previously discussed were implemented in software and run on CPUs. Utilizing CPUs for software implementations of the model are beneficial in that it only requires programming and henceforth development of the model is simplified and more flexible. It is ideal for demonstrating proof-of-concept and validating models. However, by nature, software implementations require sequential processing which is detrimental for models requiring real-time applications. FPGAs serve as an ideal solution to this drawback. This hardware-based solution is low-power and small in size and allows for faster processing. The parallel processing nature of FPGAs require a hardware description language development that although may be less flexible than software development, it allows for less overhead and faster, parallel processing in comparison to CPUs, making it ideal for such a model with complex computational mechanisms. We do not consider GPU implementations to do their typically large size and power consumption, making it

less ideal for mobile, light-weight, and low-power applications.

Over the past decade , there have been increasing interest in implementing visual saliency models on FPGA for real-time processing. Bouganis et al. [110] accelerated a saliency model proposed by Li et al. [111] which operated on the gray-scale of a single image only and utilized neuron models tuned to specific orientation and spatial locations. The differential equations used to model these neurons for computing saliency is computationally-demanding and therefore, the array of neurons with their associated dynamics were implemented on FPGA. They were able to show a speed up of more than 10× when using this parallel architecture.

Kestur et al. [112] utilized FPGA to implement a library for saliency computation based on the Itti et al. (1998) model [18]. This FPGA-based accelerator is called Streaming Hardware Accelerator with Run-time Configurability (SHARC) and showed 5× speed up to CPU-based version of the saliency model on 256 × 256 images. Others have also implemented the bottom-up, feature-based Itti et. al model on FPGA. Akselrod et al. [113] utilized their NeuFlow platform of implementing a simplified Itti et al. saliency model showing a 4× speedup on 480 × 480 images in comparison to CPU implementation. Motion was also incorporated into the model. Kim et al. [114] also implemented the model on FPGA, simplifying the normalization operation for FPGA. Their implementation interfaces with a silicon retina chip and extracts various features on 128 × 128 image. They were able to show a speedup of more than 2.5× and power reduction of more than 32× by using an FPGA implemen-

tation. Moradhasel et al. [115] designed an FPGA based saliency model and showed computation speeds of 50 million pixels per second. Similarly to Akselrod et al. [113] implementation, it also considered motion in this model. It showed a 2× speed up over state-of-the-art models at that time. Most recently, Barranco et al. [116] developed a simplified, yet more complete FPGA implementation of the saliency model incorporating motion as well as winner-take-all and inhibition of return. It also has a top-down component which modulates the final saliency map as a function of optical flow and depth. This model outperformed all previous models with respect to speed as it computed saliency maps at 180 fps for 640 × 480 resolution. Finally, other saliency models have been implemented on FPGA including the work of Bae et al. [117] where the AIM (Attention based on Information Maximization) algorithm by Bruce et al. [118] was implemented on FPGA platform for real-time processing capable of 4 million pixels per sec.

These FPGA implementation of saliency models discussed demonstrate the advantages FPGA implementations have over CPU implementation with regard to processing speed and power consumption. However, all of these models are purely feature-based. The proto-object-based saliency model of Russel et. al and our novel dynamic proto-obect based model (DPOVS-ST) discussed in Chapter 2 showed to outperform the Itti et. al model and other SOTA models in predicting human eye fixations. In this model saliency was instead computed as a function of proto-objects existing within the visual scene. The work presented here is novel two-fold:

1. We incorporate motion into the proto-object based visual saliency model

2. We implement this complete dynamic proto-object based visual saliency model on FPGA for real-time processing and to our knowledge, is the first real-time object-based visual saliency model implemented on FPGA for real-time processing. Computation of proto-objects is computationally-demanding and therefore an FPGA implementation is ideal.

3.3 Modified Dynamic Proto-object Based Visual Saliency Model

Considering the complexity of the DPOVS-ST model discussed in Chapter 2 and limited resources on the FPGA, slight modifications were made to the model for FPGA implementation. The overall flow and steps of the model remain the same. However, specifications of the model were slightly modified without compromising the biological-plausibility or basis of the model. The modifications are outlined in Table 3.1.

Table 3.1: Summary of DPOVS-ST Model Modifications for FPGA Implementation

Model Specification	**MATLAB Implementation**	**FPGA Implementation**
Kernel Size (Grouping)	11×11	5×5
# Pyramid Levels	10	3
Resolution (W × L)	640×480	112×84 (or 80×60)
Feature Channels	9	9
Precision	Floating Point	8-Bit

3.3.1 Filter Kernel Size

The first modification is to the kernel size used in the filtering tasks within the grouping mechanism stage. In the software (MATLAB) implementation, the kernel size is 11×11 for performing the various edge filtering, center-surround filtering, von mises filtering, and related tasks. However, in the FPGA implementation, the filter size is 5×5 for these tasks within the grouping mechanism stage. For performing a filtering task, the larger the kernel size the more multiply-accumulate per weighted sum. In this implementation, it takes one clock cycle, T_{period}, to perform a single multiply-accumulate. Therefore, the total time, T_{filter} for performing a filter task on a single image/frame is represented in Equation 3.1.

$$T_{filter} = T_{period} \times (K_{width} \times K_{length}) \times (N_{rows} \times N_{cols}) \tag{3.1}$$

The parameters K_{width} and K_{length} are the kernel width and length size. The parameters N_{rows} and N_{cols} are the number of rows and columns of the current image/frame that on which the filtering task is being performed. Henceforth, reduction in kernel size is necessary for increasing processing speed as less multiply-accumulate operations are required. A nearest-neighbor subsampling method is used for reducing kernel size.

3.3.2 Resolution and Pyramid Levels

The second modification to the model is the input resolution. The MATLAB implementation has an input resolution of 640×480 pixels ($W \times L$). The FPGA implementation has a reduced resolution of either 112×84 or 80×60. The reduced resolution is necessary for increasing processing speed as computations such as a filtering task is a function of the size of the input image N_{rows} and N_{cols} as seen in Equation 3.1. Furthermore, the FPGA is limited with the amount of memory (block RAM) allocated for this saliency model implementation. Due to limited block RAM on the FPGA 84×112 was the maximum input resolution. Considering the smaller resolution, the number of pyramid levels was also reduced. The reason for computing on an image pyramid is to allow for scale-invariance. The MATLAB implementation has a resolution of 640×480 at the top level of the pyramid and scales down to 30×23 at the lowest level of the pyramid. Operating at a resolution of 112×84 does not require as many levels in the image pyramid so only 3 levels sufficed.

3.4 FPGA Implementation

The FPGA implementation for real-time computation of dynamic proto-object based visual saliency will be discussed in this section in detail. The objective of this hardware, parallel architecture implementation is for fast, real-time processing of the dynamic saliency map. Therefore, some components of the model did not require a hardware implementation for fast processing. We utilize a hybrid approach in which some components of the model were implemented on software (via MATLAB) while more complex, computationally-heavy tasks were implemented on hardware (FPGA). An overall idea of the this hybrid approach is visualized in Fig. 3.2.

The initial feature extraction and motion extraction occurs on PC via MATLAB. The FPGA component is responsible for performing the border ownership computation and grouping computation. The pyramid collapsing, normalization operators , and summing across feature channels for computing the final dynamic saliency map is computed on PC. The input to the model can be either a video stored on PC or live input from an external or built-in webcam. Each component of the DPOVS-ST model for implementation on the FPGA for real-time implementation will be discussed in the proceeding sections. Before discussing these details, however, the specifications of the FPGA used will be discussed.

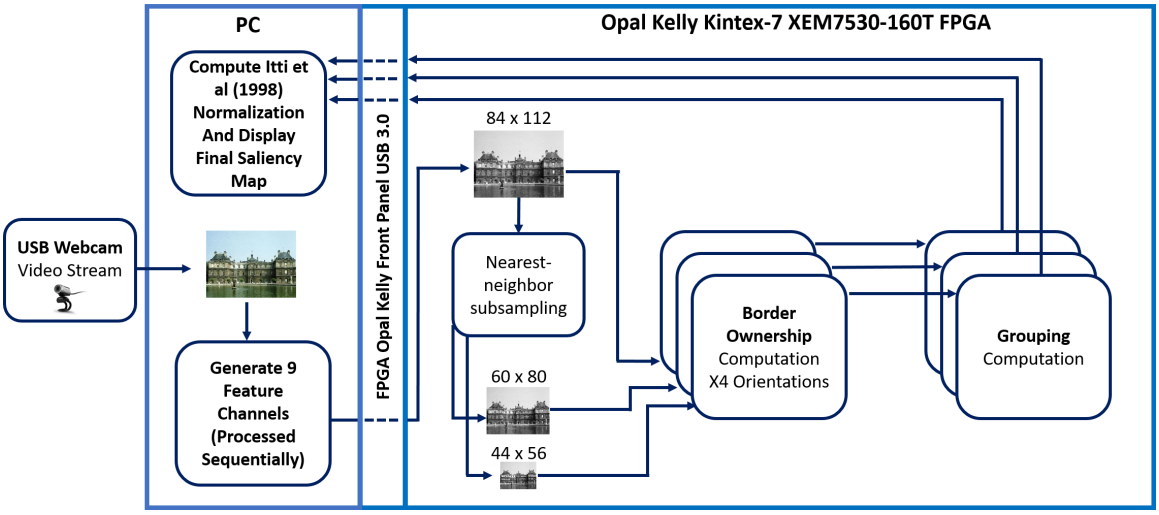


Figure 3.2: Top Level Block Diagram of FPGA-based DPOVS-ST Model. Input video stream is either generated on PC or received via USB or other webcam. Feature extraction computation occurs within MATLAB on PC. The border ownership and grouping computation occurs predominantly on FPGA. The grouping results are transmitted to PC for the pyramid collapsing and normalizations to be performed to compute final dynamic saliency map. Image from [22].

Figure 3.3: Opal Kelly XEM7350-160T Field-Programmable Gate Array

3.4.1 FPGA Specifications

The development board used in this implementation is the Opal Kelly XEM7350-160T (See Fig. 3.3). This board consists of a Xilinx Kintex-7 FPGA. It is necessary to highlight select specifications of this FPGA as these specifications presented limitations to which the model was designed around. These specifications can be seen in Table 3.2. Note that we utilize a 100 MHz clock in this implementation.

Table 3.2: Opal Kelly XEM7350-160T Kintex-7 FPGA Highlighted Specifications

Model Specification	***Value***
Clock Speed	100 MHz (or 200 MHz)
Slice Count	$25,350$
D Flip-Flops	$202,800$
Block RAM	$11,700$ Kib (1.4625 MB)
DSP Slices	600
I/O Ports	168
Physical Dimensions	$80mm \times 70mm \times 15.1mm$

The Opal Kelly FPGA series also allows for the use of their FrontPanel API. The FrontPanel API facilitates communication via USB 3.0 (or USB 2.0) between PC (MATLAB) and FPGA.

3.4.2 Model

The detailed FPGA model can be seen in Fig. 3.4 and will be discussed in this section.

3.4.2.1 Input Video Stream

The input to the model is either an RGB (color) image or video. Video input is composed of frames which can be seen as a sequence of images. The resolution expected by the model is either 112×84 or 80×60 pixels ($W \times L$). This input can be from a video stored on the PC or via a USB or built-in camera. Any input image/frame must be resized to the resolution expected by the model. The individual pixel resolution is 8-bit, and henceforth, the input image/frame is always linearly normalized between 0 to 255 prior sending through the model. This rescaling and normalization occurs in software (MATLAB).

3.4.3 Feature and Motion Extraction

3.4.3.1 Spatial Information

The next stage of the model is the feature extraction for generating the various feature channels. Within the intensity channel, the temporal information is extracted using the same method depicted in Section 2.5.1 in the previous chapter. The biologically-plausible strongly phasic temporal filter is used for extracting motion.

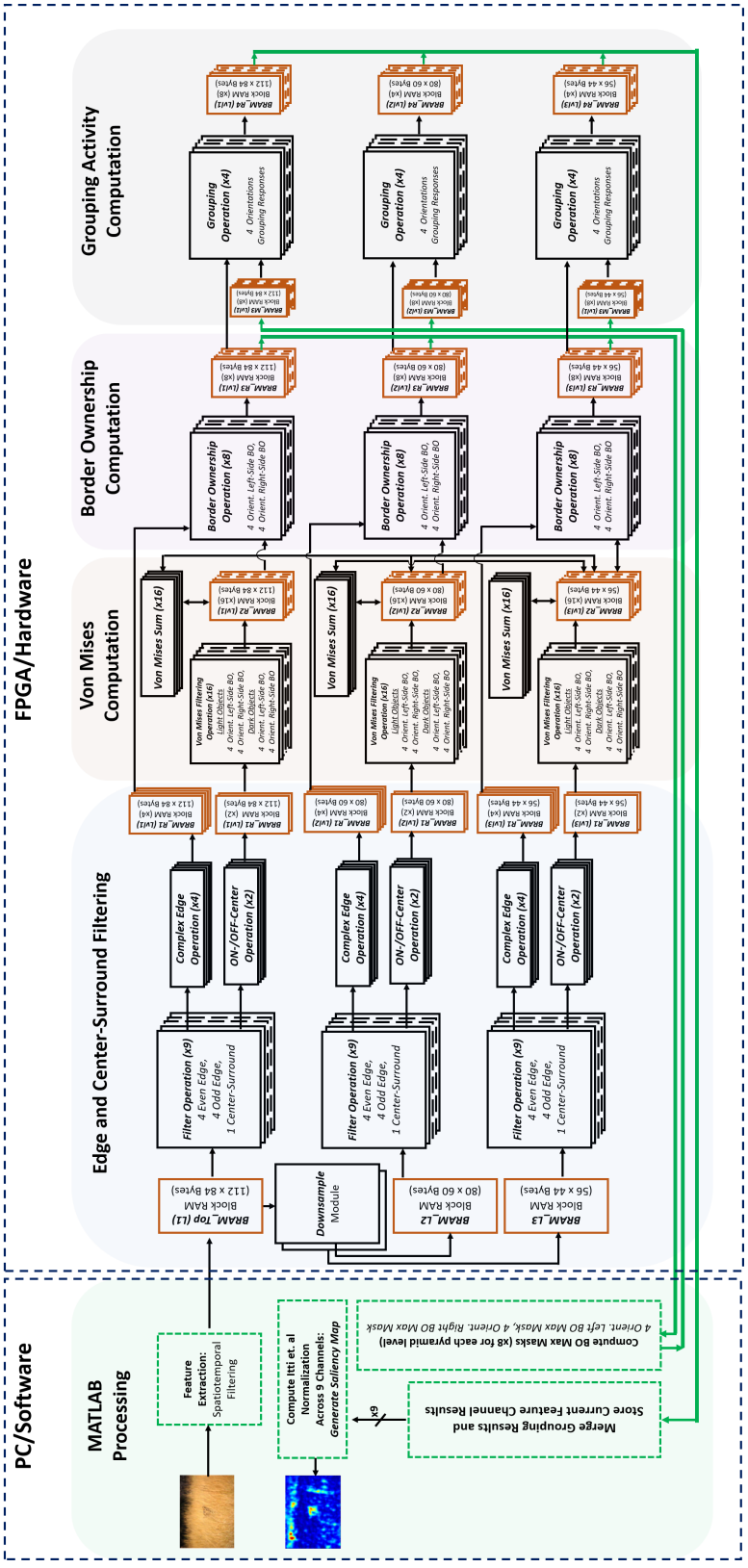


Figure 3.4: Complete FPGA-based Dynamic Proto-object Based Visual Saliency Model

The gray-scaled version of the temporal output is then used as input to the model.

For the color channel, the same 4 subchannels are extracted: red-green opponency, green-red opponency, blue-yellow opponency, and yellow-blue opponency. Prior to extracting these color opponencies, motion is extracted using the weakly phasic temporal filter. The color opponencies are then computed using the same methods in Section 2.5.1.

Similarly, to the software implementation, motion is not extracted within the orientation channel. The gray-scaled version of the current input/frame serves as input to the 4 orientation subchannels ($0, \frac{\pi}{4}$, $\frac{\pi}{2}$, and $\frac{3\pi}{4}$).

3.4.3.2 Temporal Information

The temporal filters used, both strongly phasic and weakly phasic operate over the current time and ~ 150 in the past. Assuming the input video is at a frame rate of 24 Hz, this results in a temporal convolution over the 6 frames (the current frame and 5 previous frames). This supported by Equation 3.2.

$$\texttt{\# Frames} = \frac{\texttt{Filter Duration}}{\texttt{Frame Rate}} = floor\big(\frac{150 \times 10^{-3}}{24\texttt{FPS}}\big) = 6 \ \texttt{Frames} \tag{3.2}$$

For storing these 5 prior frames, we utilize memory allocated within MATLAB. Considering the resolution of the image, storing these 5 frames within MATLAB and extracting motion on these 6 frames within MATLAB was sufficient. As each new frame is received, the 6^{th} frame (furthest in the past) is removed and each frame

shifts down by one frame in the allocated memory (See Fig. 3.5. MATLAB efficiently

```
%Get new motion frames
motion_im(:,:,:,1) = motion_im(:,:,:,2);
motion_im(:,:,:,2) = motion_im(:,:,:,3);
motion_im(:,:,:,3) = motion_im(:,:,:,4);
motion_im(:,:,:,4) = motion_im(:,:,:,5);
motion_im(:,:,:,5) = motion_im(:,:,:,6);
motion_im(:,:,:,6) = im;
motion_out_s  = computeTemporalFiltering(
```

Figure 3.5: Snippet of MATLAB code for continuous storage and shifting of frames of video input. motion_im is memory allocated for storing 6 frames while im is the current frame.

performs array and vector-based computation, allowing for the temporal convolution within MATLAB to occur with low computational cost with regards to resources and time. It is merely a multiplication of six unique constant values with six unique frames, respectively, followed by a summation of these results. The absolute value of these results represent the motion output at the current frame. This can be seen in Equation 3.3. Note that this shows only the strongly phasic coefficients and computation for motion extraction, however, the same applies for the weakly phasic filter.

$$\texttt{Strongly Phasic Motion} = s_1 * F_t + s_2 * F_{t-1} + s_3 * F_{t-2} + s_4 * F_{t-3} + s_5 * F_{t-4} + s_6 * F_{t-5} \tag{3.3}$$

The parameters $s_1, ..., s_6$ is the strongly phasic temporal filter coefficients. The values of these coefficients can be found in Table 3.3. The parameters $w_1, ..., w_6$ are the coefficients for the weakly phasic temporal filter. F is such that F_t is the current

frame, F_{t-1} is one frame in the past, F_{t-2} is two frames in the past, etc. The time for storing video frames and performing a single temporal filter on 6 frames is ~ 2.1 ms. A total of $\sim 170KB$ of memory is necessary for allocating space for storing 6 frames of RGB color frames/images.

Table 3.3: Strongly Phasic and Weakly Phasic Filter Coefficient Values Over 6 Frames

Frame (f)	***Strongly Phasic Coeff.*** (s_f)	***Weakly Phasic Coeff.*** (w_f)
F_t	$s_1 = -0.0400$	$w_1 = -0.0026$
F_{t-1}	$s_2 = -0.0403$	$w_2 = -0.0326$
F_{t-2}	$s_3 = -0.0974$	$w_3 = -0.0275$
F_{t-3}	$s_4 = -0.2051$	$w_4 = +0.1797$
F_{t-4}	$s_5 = +0.3792$	$w_5 = +0.1319$
F_{t-5}	$s_6 = +0.0035$	$w_6 = +0.01586$

3.4.4 Grouping Computation

The grouping computation occurs predominantly on the FPGA. Grouping is computed within each of the nine feature channels independently and sequentially. In this section we describe the method for computing the grouping activity within a single channel. The steps are outlined below.

1. P1: Transmit Spatiotemporal Feature Extraction Output

2. P2: Generate Frame Pyramid

3. P3: Complex Edge and Center-Surround Filtering

4. P4: Von Mises Filtering

5. P5: Von Mises Sum

6. P6: Border Ownership Responses

7. P7: Grouping Responses

3.4.4.1 P1: Transmit Spatiotemporal Feature Extraction Output

The first step involves transferring the output of the spatiotemporal feature extraction (within one of the nine feature channels) to the FPGA for processing. This output is the result of the motion extraction (within intensity and color channels) and spatial feature channel. It is a single frame of size 84×112 pixels. Each pixel has a resolution of 8-bits. This data is transferred to block RAM on the FPGA. Therefore, the total amount of data transferred and size of this block RAM ($BRAM\_Top$) is ~ 75 KB and is a function of the resolution which can be seen in Equation 3.4.

$$\texttt{Transmitted Data (Bits)} = N_{rows} \times N_{cols} \times \texttt{8-bits} = 84 \times 112 \times \texttt{8-bits} = 75.264\ \texttt{KB} \tag{3.4}$$

Each address in block RAM corresponds to a (row,col) location as seen in Equation 3.5, such that $row \in [1, ..., 84]$ and $col \in [1, ..., 112]$.

$$\texttt{BRAM Address(row,col)} = (row - 1) \times 112 + (col - 1) \tag{3.5}$$

The USB 3.0 communication allows for a transmission speed of 340 MB/s. Transmitting this frame from PC/MATLAB to FPGA takes a total time of ~ 1.7 ms.

3.4.4.2 P2: Generate Frame Pyramid

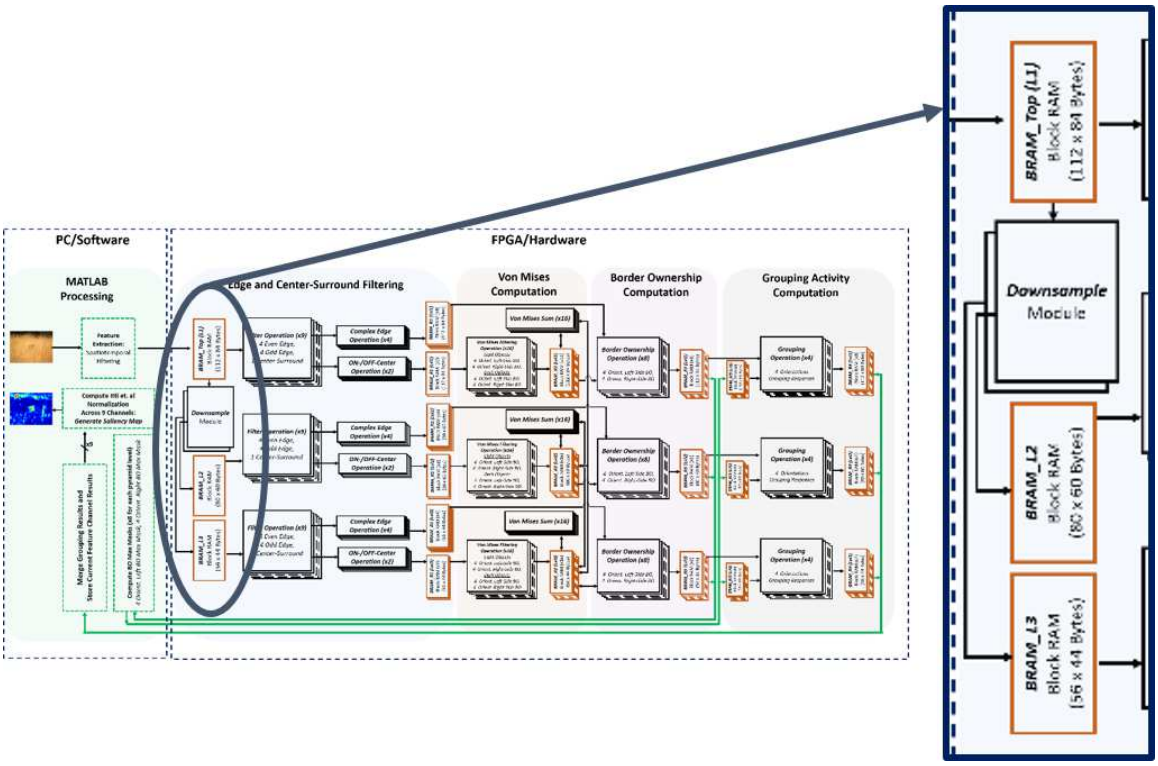


Figure 3.6: Pyramid Generation Component of FPGA Model

The next stage in processing is the generation of the frame/image pyramid (See

Fig. 3.6). This model consists of 3 pyramid level and therefore, 2 additional images must be generated. The two additional levels have a resolution of 80×60 and 56×44, respectively. A "Downsample" module is used for generating these two images. This module uses a "nearest-neighbor" type downsampling implementation. Given the scaling factor with respect to the resolution of the top level, Equation 3.6 can be used for creating the new level in the pyramid.

$$\begin{aligned} Row_{new} &= \alpha_{row}(l) * row \\ Col_{new} &= \alpha_{col}(l) * col \\ \alpha_{row}(1) &= \frac{5734}{4096} \approx \sqrt{2}^{1} \\ \alpha_{row}(2) &= \frac{7820}{4096} \approx \sqrt{2}^{2} \\ \alpha_{col}(1) &= \frac{5734}{4096} \approx \sqrt{2}^{1} \\ \alpha_{col}(2) &= \frac{8192}{4096} \approx \sqrt{2}^{2} \end{aligned} \tag{3.6}$$

For each level (l) in the pyramid, a state machine module, *Downsample*, is used to generate and store the new pyramid level. For each *row* and *col* in the pyramid, Equation 3.6, is used for determining the pixel value at the (*row*,*col*) location. Assuming I_l where I is the image and l is the level in the pyramid (I_1 is the top level of the pyramid), the pixel values at image I_l (or I_{level}) can be determined as seen in Equation 3.7.

$$I_{level}(row, col) \leftarrow I_1(Row_{new}, Col_{new}) \tag{3.7}$$

The scaling factors α_{row} and α_{col} consist of a divisor which is selected as a power of 2 such that the Row_{new} and Col_{new} can be computed by a multiplication followed by a simple right bit shift. The precision of the ratio is 12 bits, resulting in a division of $2^{12} = 4096$ (or right bit shift by 12 bits).

Considering that subsampling for each level in the pyramid requires access to the top level block RAM ($BRAM\_Top$), subsampling is computed sequentially for each pyramid level. A summary of the time for computation and additional block RAM necessary for generating the pyramid can be seen in Table 3.4.

Table 3.4: Generating Pyramid Specifications (100 MHz Clock)

Model Specification	***Value***
Clock Cycles - Level 2	24000 cycles (240.0 μs)
Clock Cycles - Level 3	12320 cycles (123.2 μs)
Additional BRAM - Level 2	4.800 KB
Additional BRAM - Level 3	2.464 KB
Total Clock Cycles	36320 cycles (363.2 μs)
Total Additional BRAM	7.264 KB

3.4.4.3 P3: Complex Edge and Center-Surround Filtering

The next stage in computation involves performing parallel filtering tasks with supporting computation for extracting the complex edge response and both ON-

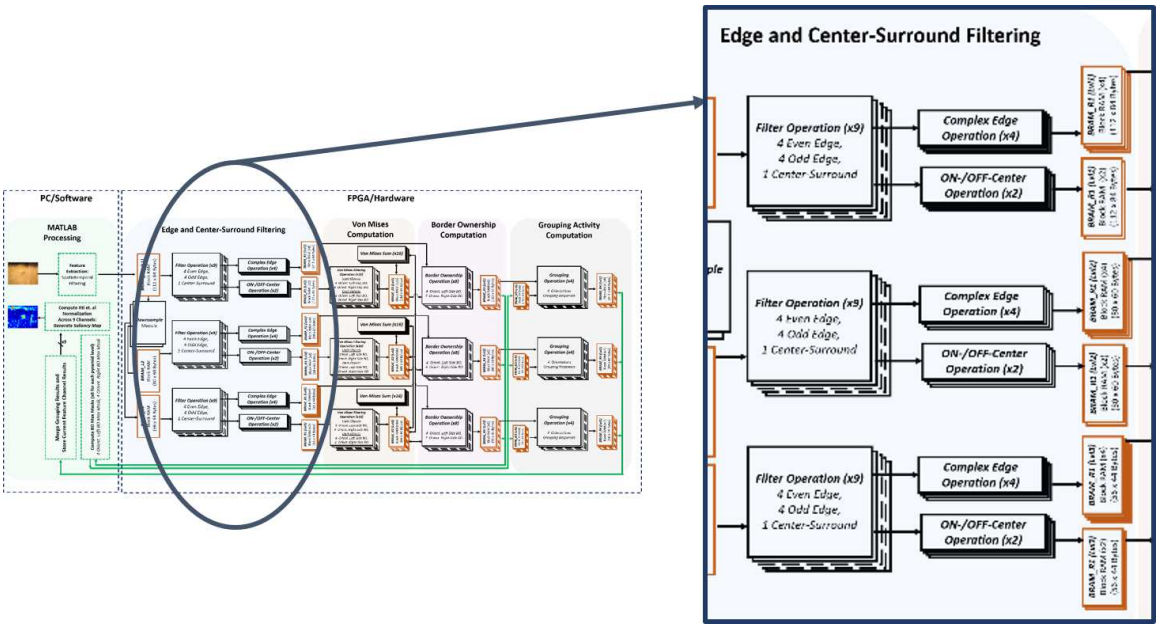


Figure 3.7: Edge Filtering (4 Orientations) and ON-Center and OFF-center Filtering Component of FPGA Model

center-surround and OFF-center-surround responses within the current feature channel (See Fig. 3.7. The complex edge responses are extracted by computing the square root of the even and odd edge responses. Even and odd edge filter kernels are computed as depicted in the previous chapter (Equations 2.16 and 2.17). The parameters of these kernels are selected such that there are 4 in a 5×5 even edge kernels and 4 5×5 odd edge kernels (See Fig. 3.8). The center-surround kernels are explained in detail in the previous chapter. The Equations 2.20 and 2.21 depict the ON- and OFF-center-surround kernels for non-Orientation feature channels. The Equations 2.22 and 2.22 depict the ON- and OFF-center-surround kernels for the Orientation channels. The parameters are also selected such that a single ON-center-surround

kernel is used for both the Orientation channels and non-Orientation channels and the OFF-center-surround response is simply the inverted response of the ON-center-surround response. The resulting 5×5 center-surround kernels can be seen in Fig. 3.8.

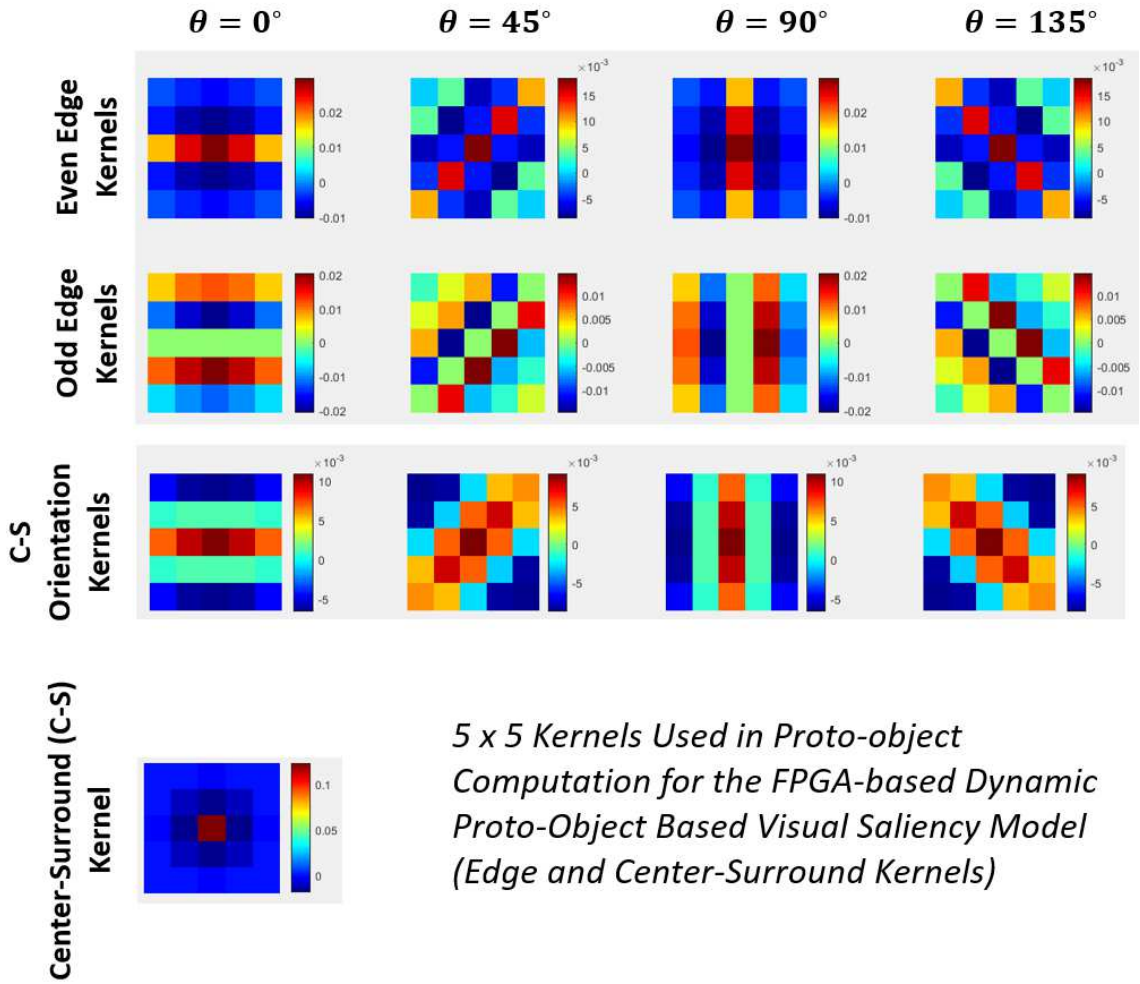


Figure 3.8: Edge (even and odd) and center-surround kernels (5×5) used in the FPGA-based DPOVS-ST model. Edge kernels for all four orientations are shown. Center-surround kernels for non-orientation channels and the 4 center-surround kernels for the 4 orientation subchannels are shown.

For computing these complex edge and center-surround responses, a finite state machine (FSM) is used which performs the following steps:

1. Load 5×5 patch at the current pixel location
2. Compute weighted-sum for 8 edge and ON-center-surround response
3. Compute square root for complex edge response of all 4 orientations
4. Invert ON-center-surround response for OFF-center-surround response
5. Resulting 6 responses are stored in BRAM

Note that this process occurs for each pixel in the image and is computed in parallel at each pyramid level.

3.4.4.3.1 Load Image Patch

This module consists of a state machine for loading a 5×5 image patch containing the neighboring pixels of the current pixel location on which the weighted-sum is being computed. Based on the current (row, col), each address from ($row - 2$, $col - 2$) to ($row + 2$, $col + 2$) is obtained and the corresponding location in BRAM is computed using Equation 3.8 where N_{cols} is the number of columns in the current image.

$$\texttt{BRAM Address(row,col)} = (row - 1) \times N_{cols} + (col - 1) \tag{3.8}$$

This address is then used for reading the 8-bit pixel value at that location. This 8-bit value is then stored in a 5×5 8-bit register array. The number of clock cycles required for loading this image patch is seen in Equation 3.9.

$$\texttt{\# Clock Cycles} = K_{width} \times K_{length} \times 5 = 125 \texttt{ Clock Cycles} \tag{3.9}$$

The parameters $K_{width} = 5$ and $K_{length} = 5$ and 5 clock cycles are required for loading and storing a single pixel value. Therefore, the total clock cycles for loading the image patch is 125 cycles. Table 3.5 summarizes the time required for loading an image patch.

Table 3.5: Loading a 5×5 Image Patch (100 MHz Clock)

Model Specification	**Value**
Total Clock Cycles	125 cycles (1.25 μs)

3.4.4.3.2 Compute Weighted-Sum

The next step in the filtering module is computing the weighted-sum. Following the loading of the image patch, an additional state-machine is used for sequentially computing the weighted-sum of the 5×5 image patch with the corresponding 5×5 kernel. The weighted sum mathematically can be expressed as following:

$$\texttt{Weighted Sum} = \sum_{r=1}^{5}\sum_{c=1}^{5} K(r,c) \times P(r,c) \tag{3.10}$$

The values r and c are the row and column, respectively, of the 5×5 kernel (K) and 5 image patch (P). This is implemented in hardware as a series of multiply-accumulates (MAC). The weighted-sum is essentially a sequences of MACs to obtain the final result at each pixel location. For each of the 4 even edge kernels, 4 odd edge kernels, and ON-center-surround kernel, the weighted-sums are computed in parallel. Each of these 9 weighted-sum operations occurring in parallel consists of a single MAC module each. The number of clock cycles required for performing a single MAC operation is 3 cycles. Furthermore, it requires 25 MAC operations (75 clock cycles) to compute the weighted-sum results for all 9 kernels.

$$MAC_{WeightedSum} = Time_{MAC} \times K_{width} \times K_{length} = 3 \times 5 \times 5 = 75 \texttt{ Clock cycles} \tag{3.11}$$

Each kernel value, $K(r,c)$, has a precision of 12 bits and can be either positive or negative, hence, requiring 13 bits for each kernels value (for all 9 kernels). Therefore, the multiplication of the 8-bit pixel value by the kernel value occurs as seen in Equation 3.12.

$$\texttt{Multiplication Result} = \frac{P(r,c) \times \texttt{[13-bit Kernel Value]}}{2^{12}} \tag{3.12}$$

Note that the result is always then normalized by 2^{12} considering the 12-bit precision used in the kernel value. This is necessary for obtaining a result back in its 8-bit precision representation.

Table 3.6: Weighted Sum (100 MHz Clock)

Model Specification	**Value**
Total Clock Cycles	75 cycles (0.75 μs)

3.4.4.3.3 Computing Complex Edge and ON-/OFF-Center-Surround

Following the weighted sum computation for the current image patch, the complex edge responses and ON- and OFF-center-surround responses are computed for the current pixel location. There are four complex edge responses which are computed for four oriented edges (0, $\frac{\pi}{4}$, $\frac{\pi}{2}$, and $\frac{3\pi}{4}$). The complex edge is computed by taking the square root of the even edge response ($S_e(r,c)$) squared summed with the odd edge response ($S_o(r,c)$) squared as seen in Equation 3.13.

$$\texttt{Complex Edge Response} = \sqrt{S_e(r,c)^2 + S_o(r,c)^2} \tag{3.13}$$

This complex edge response for each pixel requires 3 clock cycles. One clock cycle for squaring both the even and edge response, one clock cycle for summing these results, and a final clock cycle for computing the square root of this sum. An IP Core Generator is used for generating the square root modules which run on the DSP

blocks of the FPGA. For each image, 4 square root computations occur in parallel for computing the 4 complex edge responses in parallel.

For computing the ON- and OFF-center-surround responses, the following method is used. The ON-center-surround result obtained from the weighted-sum using the center-surround kernel is used for determining the ON- and OFF-center-surround response. Given the center-surround (CS) weighted-sum (ws) response, $CS_{ws}(r, c)$, the ON-center and OFF-center responses can be computed as seen in Equations 3.14 and 3.15. For the ON-center response, if the sign bit is positive ("0"), the lower 8-bits are used as the result. If the sign bit is negative ("1"), the result is rectified to zero. Similarly, if the sign bit is negative ("1"), the two's complement is used as the result (for obtaining the magnitude of the response) and if the sign bit is positive ("0"), the result is rectified to zero.

$$\texttt{ON-Center Response}(r,c) = \begin{cases} CS_{ws}(r,c) & \texttt{,if sign-bit = 0} \\ 0 & \texttt{,if sign-bit = 1} \end{cases} \tag{3.14}$$

$$\texttt{OFF-Center Response}(r,c) = \begin{cases} \texttt{Two's Complement}\big(CS_{ws}(r,c)\big) & \texttt{,if sign-bit = 1} \\ 0 & \texttt{,if sign-bit = 0} \end{cases} \tag{3.15}$$

Computing the two's complement is a combinatorial process. Therefore, only

a single (1) clock cycle is required for extracting the ON-center and OFF-center-surround responses from the $CS_{ws}(r, c)$ result.

The total clock cycles required for extracting the complex edge responses and center-surround responses is the max clock cycles between complex edge extraction and center-surround extraction. The complex edge responses require 3 clock cycles opposed to the center-surround responses requireing 1 clock cycle. Therefore, a total of 3 clock cycles is required after the weighted-sum is computed. The time required for extracting the 4 complex edge responses and 2 center-surround responses is summarized in Table 3.7.

Table 3.7: Complex Edge and Center-Surround Responses (100 MHz Clock)

Model Specification	**Value**
Total Clock Cycles	3 cycles (30 ns)

3.4.4.3.4 Store Results in BRAM

The final step is a parallel write to 6 different BRAMS for storing the results (4 complex edge responses and ON-center and OFF-center surround responses). The address in block RAM is computed in parallel with the previous operations. This results in computing the address to store the result at no cost in time. The writing of these 6 responses to BRAM requires only 1 clock cycle (See Table 3.8).

Table 3.8: Storing Responses in BRAM (100 MHz Clock)

Model Specification	Value
Total Clock Cycles	1 cycle (10 ns)

3.4.4.3.5 Summary of Time and Resources for Filtering Task

Note that the steps for computing complex edge responses and ON-/OFF-center-surround responses explained in this section is for a single pixel location. These responses must be computed for each pixel location in the image and is a sequential process. Furthermore, this operation occurs in parallel for each of the three levels in the image pyramid. The time required for computing the 4 complex edge responses and 2 center-surround responses across the entire image for a single pyramid level can be computed as seen in Equation 3.16.

$$\begin{aligned}\texttt{Total Time (level)} &= (T_{loadpatch} + T_{ws} + T_{ResFin} + T_{store}) \times N_{rows}(level) \times N_{cols}(level) \\ &= (125\ \texttt{CC} + 75\ \texttt{CC} + 3\ \texttt{CC} + 1\ \texttt{CC}) \times N_{rows}(level) \times N_{cols}(level) \\ &= 204\ \texttt{Clock Cycles (CC)} \times N_{rows}(level) \times N_{cols}(level)\end{aligned} \tag{3.16}$$

The total additional BRAM for storing the 6 responses (of equal size of the original image) can be computed as seen in Equation 3.17

$$\begin{aligned} \texttt{Additional BRAM (level)} &= Precision \times N_{rows}(level) \times N_{cols}(level) \times 6 \\ &= 8 \texttt{ bits} \times N_{rows}(level) \times N_{cols}(level) \times 6 \end{aligned} \tag{3.17}$$

Table 3.9 summarizes the time and additional BRAM required for performing these filtering tasks and obtaining these 6 responses for each pyramid level.

Table 3.9: Summary of Time and BRAM Required for Filter Responses (×6 → 4 Complex Edge, 1 ON-Center, 1 OFF-Center Response) (100 MHz Clock)

Model Specification	***Value***
Time (Level 1, 6 Responses)	1,919,232 CC (19.2ms)
Time (Level 2, 6 Responses)	979,200 CC (9.8ms)
Time (Level 3, 6 Responses)	502,656 CC (5.0ms)
Additional BRAM (Level 1, 6 Responses)	56,448 Bytes
Additional BRAM (Level 2, 6 Responses)	28,800 Bytes
Additional BRAM (Level 3, 6 Responses)	14,784 Bytes
Total Time (All Levels, 18 Responses)	1,919,232 CC (19.2ms)
Additional BRAM (All Levels, 18 Responses)	100.032 KB

3.4.4.4 P4: Von Mises Filtering

The next stage of processing uses the output from the center-surround responses for both ON-center-surround (light objects on dark background) and OFF-center-

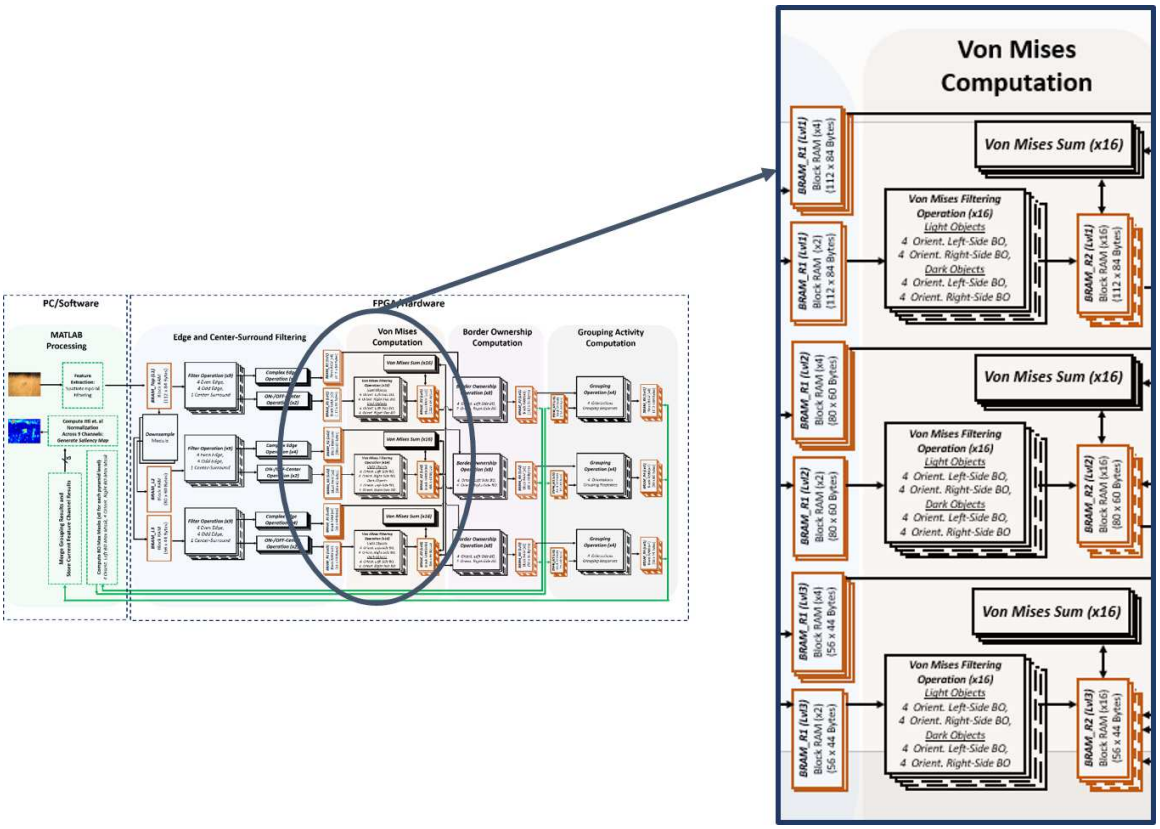


Figure 3.9: Von Mises Filtering and Summing Component of FPGA Model

surround (dark objects on light background) responses (for each pyramid level) to compute the von mises response (See Fig. 3.9). The von mises response is necessary for mapping edges to their corresponding center-surround responses (objects). The von mises responses are then used for computing the border ownership responses for both left and right sides of the oriented border. To reiterate, border ownership response for light objects and dark objects can be seen in Equations 3.18 and 3.19, respectively.

$$B^k_{\theta,L}(x,y) = \left\lfloor C^k_\theta(x,y) \times \left(1 + \sum_{j \geq k} \frac{1}{2^j} v_{\theta+\pi}(x,y) * CS^j_L(x,y) - w_{opp} \sum_{j \geq k} \frac{1}{2^j} v_\theta(x,y) * CS^j_D(x,y) \right) \right\rfloor \tag{3.18}$$

$$B^k_{\theta,D}(x,y) = \left\lfloor C^k_\theta(x,y) \times \left(1 + \sum_{j \geq k} \frac{1}{2^j} v_{\theta+\pi}(x,y) * CS^j_D(x,y) - w_{opp} \sum_{j \geq k} \frac{1}{2^j} v_\theta(x,y) * CS^j_L(x,y) \right) \right\rfloor \tag{3.19}$$

The parameter w_{opp} is the synaptic weight of the inhibitory signal from the opposite polarity CS response ($w_{opp} = 1$). The term 2^{-j} is a factor applied such that influence across spatial scales is constant. The v_θ term is the von mises distribution-based kernel. The variable $C^k_\theta(x,y)$ is the complex edge response (of orientation θ) at pyramid level k. The variables $CS^j_L(x,y)$ and $CS^j_D(x,y)$ are the center surround responses for light and dark objects, respectively, at pyramid level j.

This stage of the model computes the filtering tasks seen in Equations 3.20 where "*" is the correlation operator. The von mises filter is convolved with the ON-center-surround response (for each pyramid level) for both left and right border ownership (v_θ

and $v_{\theta+\pi}$) von mises filters. This same process occurs for the OFF-center response (for each pyramid level). For each pyramid level, this results in 4 responses (4 different orientations:0, $\frac{\pi}{4}$, $\frac{\pi}{2}$, and $\frac{3\pi}{4}$) for left-side (θ) responses for light objects (Equation 3.20. There are 4 responses for right-side ($\theta + \pi$) responses for light objects. These 8 results are also computed for dark objects (Equation 3.21). This totals to 16 von mises responses (or resulting maps) for each pyramid level ($\times 3$).

$$\begin{aligned} VonResLight_{\theta}(k) &= v_{\theta}(x,y) * CS_L^k(x,y) \rightarrow \texttt{Left-Side Mapping} \\ VonResLight_{\theta+\pi}(k) &= v_{\theta+\pi}(x,y) * CS_L^k(x,y) \rightarrow \texttt{Right Side Mapping} \end{aligned} \tag{3.20}$$

$$\begin{aligned} VonResDark_{\theta}(k) &= v_{\theta}(x,y) * CS_D^k(x,y) \rightarrow \texttt{Left-Side Mapping} \\ VonResDark_{\theta+\pi}(k) &= v_{\theta+\pi}(x,y) * CS_D^k(x,y) \rightarrow \texttt{Right Side Mapping} \end{aligned} \tag{3.21}$$

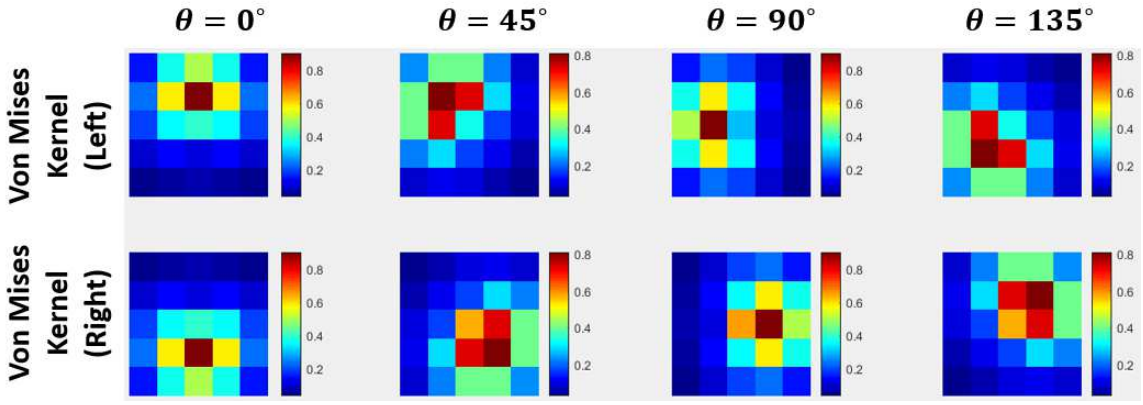


Figure 3.10: Von Mises kernels (5×5) used in the FPGA-based DPOVS-ST model. Both left (θ) and right ($\theta + \pi$) sided kernels for all four orientations are shown.

The same filtering process as seen in the previous (Section 3.4.4.3) is used for performing these 16 operations at each of the 3 pyramid levels. The loading of the image patch (125 clock cycles), weighted-sum (75 clock cycles), and storing in BRAM (1 clock cycle) operations are used with the appropriate von-mises kernels (See Fig. 3.10). For each pyramid level, the time for computing the 16 responses (occurring in parallel) can be seen in Equation 3.22. The total additional BRAM required for storing the results for this stage (for each pyramid level) can be seen in Equation 3.23.

$$\begin{aligned} &\texttt{Time for Von Mises Filtering Task (level)} \\ &= (T_{loadpatch} + T_{ws} + T_{store}) \times N_{rows}(level) \times N_{cols}(level) \\ &= (125\ \texttt{CC} + 75\ \texttt{CC} + 1\ \texttt{CC}) \times N_{rows}(level) \times N_{cols}(level) \\ &= 201\ \texttt{Clock Cycles (CC)} \times N_{rows}(level) \times N_{cols}(level) \end{aligned} \tag{3.22}$$

$$\begin{aligned} &\texttt{Additional BRAM (level)} \\ &= Precision \times N_{rows}(level) \times N_{cols}(level) \times 16 \\ &= 8\ \texttt{bits} \times N_{rows}(level) \times N_{cols}(level) \times 16 \end{aligned} \tag{3.23}$$

All of these filtering tasks occur in parallel and results are stored in 48 independent block RAM modules (16 for each pyramid level). The total time and additional block RAM required for this von mises computation can be seen in Table 3.10.

Table 3.10: Summary of Time and BRAM Required for Von Mises Responses (×16 → 4 Light Objects/Left-Side Responses, 4 Light Objects/Right-Side Responses, 4 Dark Objects/Left-Side Responses, 4 Dark Objects/Right-Side Responses), (100 MHz Clock)

Model Specification	***Value***
Time (Level 1, 16 Responses)	1,891,008 CC (18.9ms)
Time (Level 2, 16 Responses)	964,800 CC (9.6ms)
Time (Level 3, 16 Responses)	495,264 CC (5.0ms)
Additional BRAM (Level 1, 16 Responses)	150,528 Bytes
Additional BRAM (Level 2, 16 Responses)	76,800 Bytes
Additional BRAM (Level 3, 16 Responses)	39,424 Bytes
Total Time (All Levels, 48 Responses)	1,891,008 CC (18.9ms)
Additional BRAM (All Levels, 48 Responses)	266.752 KB

3.4.4.5 P5: Von Mises Sum

The next stage of processing involves the von mises summing across pyramid levels (See Equation 3.24). This can be seen in Fig. 3.9. A state-machine is used for performing this summation for each pyramid level, k, sequentially. There exists 16 state machines occur in parallel (8 for von mises responses for light objects and 8 for von mises responses for dark objects from previous section).

$$\begin{aligned}
\texttt{Von Mises Sum (Light/Left)} &= \sum_{j \geq k} \frac{1}{2^j} v_\theta(x,y) * CS_L^j(x,y) \\
\texttt{Von Mises Sum (Light/Right)} &= \sum_{j \geq k} \frac{1}{2^j} v_{\theta+\pi}(x,y) * CS_L^j(x,y) \\
\texttt{Von Mises Sum (Dark/Left)} &= \sum_{j \geq k} \frac{1}{2^j} v_\theta(x,y) * CS_D^j(x,y) \\
\texttt{Von Mises Sum (Dark/Right)} &= \sum_{j \geq k} \frac{1}{2^j} v_{\theta+\pi}(x,y) * CS_D^j(x,y)
\end{aligned} \tag{3.24}$$

For each of the parallel 16 processes, the following steps occur:

1. Apply scaling factor to obtain pixel value in other pyramid level (3 clock cycles)
2. Retrieve pixel value in bottom adjacent level (1 clock cycle)
3. Multiply by factor 2^{-j} where j is the pyramid level (1 clock cycle)
4. Accumulate result for current pyramid level (1 clock cycle)
5. Repeat prior steps for each lower pyramid level, k such that $\rightarrow k \leq j$
6. Repeat all previous steps for each pyramid level j

7. Store results in BRAM used for von mises filtering (replace values) (1 clock cycle)

Note that no additional block RAM is required as the results of the von mises sum replace the values in the von mises filtering responses as they are no longer required for the remaining of the model. The time required for computing the von mises sum for each of the 16 parallel summations (for all 3 levels) can be seen in Equation 3.25. The process previously outlined must occur for each pixel location in the image/map.

$$\begin{aligned}
&\texttt{Time Required for VM Summing} \\
&= \sum_{j=1}^{3}\sum_{k=j}^{3}(3\ \texttt{CC} + 1\ \texttt{CC} + 1\ \texttt{CC} + 1\ \texttt{CC}) \times N_{rows}(j) \times N_{cols}(j) \\
&= 6\ \texttt{CC} \times 3 \times (84 \times 112) + 6\ \texttt{CC} \times 2 \times (60 \times 80) + 6\ \texttt{CC} \times 1 \times (44 \times 56) \\
&= 169344\ \texttt{CC} + 57600\ \texttt{CC} + 14784\ \texttt{CC} \\
&= 241,728\ \texttt{Clock Cycles (CC)}
\end{aligned} \tag{3.25}$$

Table 3.11 summarizes the additional BRAM required and time required for computing the von mises summation for all 16 parallel operations for each of the 3 pyramid levels.

3.4.4.6 P6: Border Ownership Responses

The next stage in processing is the border ownership responses which is computed using Equations 3.18 and 3.19 (border ownership for light objects and dark objects,

Table 3.11: Summary of Time and BRAM Required for All 16 Von Mises Responses for Each of the 3 Pyramid Levels (×48) (100 MHz Clock)

Model Specification	Value
Total Time (All Levels, 48 Responses)	241,728 CC (2.4ms)
Additional BRAM (All Levels, 48 Responses)	0.0 KB

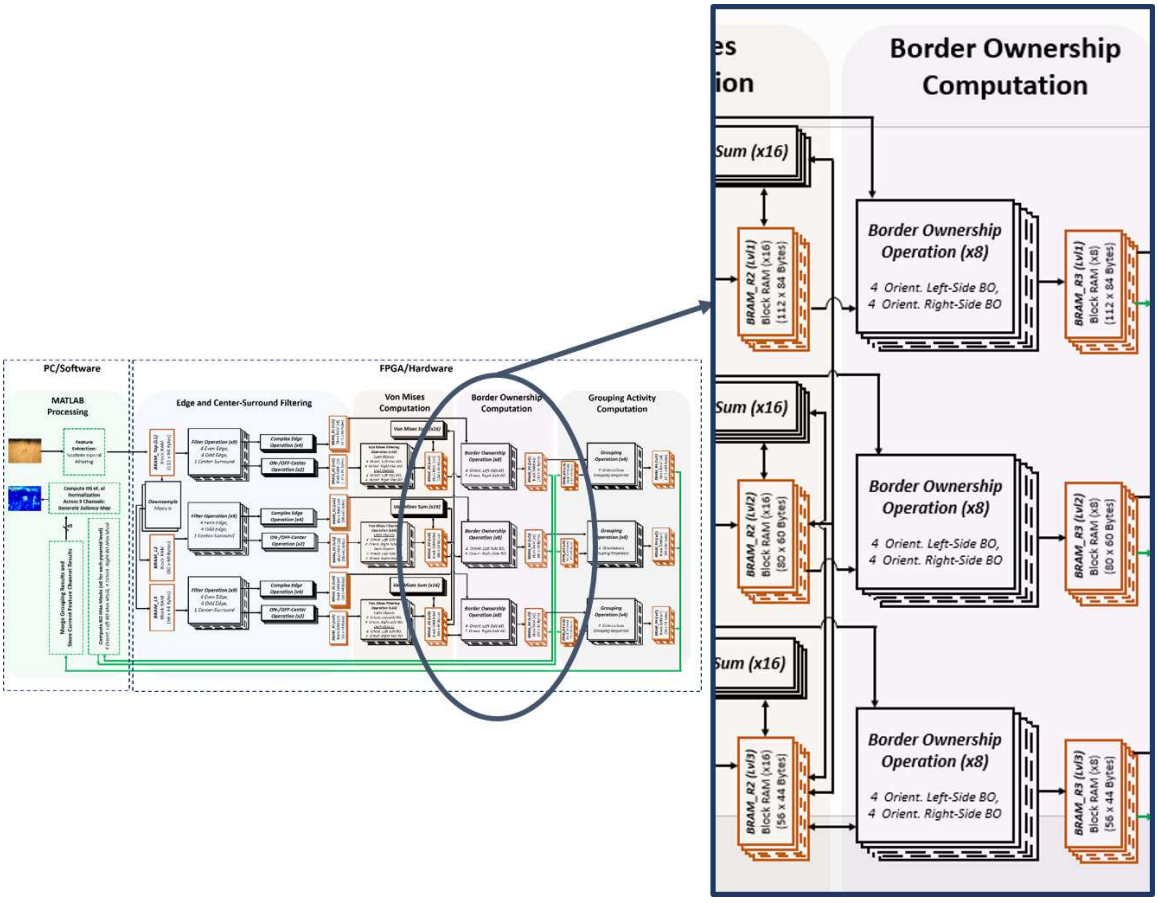


Figure 3.11: Border Ownership Computation Component of FPGA Model

respectively) for both left and right-side border ownership (θ and $\theta + \pi$). This can be seen in Fig. 3.11. The left and right border ownership responses for 4 oriented edges/borders (θ) are computed using Equations 3.26 and 3.27. They are computed by summing the border ownership response for light objects and dark objects for each oriented edge independently (polarity-invariance of center-surround response).

$$B^k_{Left}[\theta] = B^k_{Light,Left}[\theta] + B^k_{Dark,Left}[\theta] \tag{3.26}$$

$$B^k_{Right}[\theta] = B^k_{Light,Right}[\theta] + B^k_{Dark,Right}[\theta] \tag{3.27}$$

The responses $B^k_{Light,Left}[\theta]$, $B^k_{Light,Right}[\theta]$, $B^k_{Dark,Left}[\theta]$, and $B^k_{Dark,Right}[\theta]$ are computed using the von mises summation responses as seen in Equations 3.28.

$$B^k_{Light,Left}[\theta] = \left\lfloor C^k_\theta \times \left(1 + \texttt{Von Mises Sum (Light/Left)}_{k,\theta} - w_{opp} \times \texttt{Von Mises Sum (Dark/Right)}_{k,\theta}\right)\right\rfloor \tag{3.28}$$

$$B^k_{Light,Right}[\theta] = \left\lfloor C^k_\theta \times \left(1 + \texttt{Von Mises Sum (Light/Right)}_{k,\theta} - w_{opp} \times \texttt{Von Mises Sum (Dark/Left)}_{k,\theta}\right)\right\rfloor \tag{3.29}$$

$$B^k_{Dark,Left}[\theta] = \left\lfloor C^k_\theta \times \left(1 + \texttt{Von Mises Sum (Dark/Left)}_{k,\theta} - w_{opp} \times \texttt{Von Mises Sum (Light/Right)}_{k,\theta}\right)\right\rfloor \tag{3.30}$$

$$B^k_{Dark,Right}[\theta] = \left\lfloor C^k_\theta \times \left(1 + \texttt{Von Mises Sum (Dark/Right)}_{k,\theta} - w_{opp} \times \texttt{Von Mises Sum (Light/Left)}_{k,\theta} \right) \right\rfloor \tag{3.31}$$

There are 8 state machines used for each pyramid level ($\times 3$) for computing the two border ownership responses (for left border ownership and right border ownership) at each orientation for each pyramid level. Each of these state machines follow these steps:

1. Update address of current pixel location on which to perform computation (1 clock cycle)

2. Compute computation inside parenthesis for each of the 4 border ownership responses (1 clock cycle):

   $Temp11 = 1 + \texttt{Von Mises Sum (Light/Left)}_{k,\theta} - \texttt{Von Mises Sum (Dark/Right)}_{k,\theta}$

   $Temp12 = 1 + \texttt{Von Mises Sum (Light/Right)}_{k,\theta} - \texttt{Von Mises Sum (Dark/Left)}_{k,\theta}$

   $Temp21 = 1 + \texttt{Von Mises Sum (Dark/Left)}_{k,\theta} - \texttt{Von Mises Sum (Light/Right)}_{k,\theta}$

   $Temp22 = 1 + \texttt{Von Mises Sum (Dark/Right)}_{k,\theta} - \texttt{Von Mises Sum (Light/Left)}_{k,\theta}$

3. Each of these responses are multiplied by the appropriate complex edge response (1 clock cycle):

   $Temp11(x,y) = C^k_\theta(x,y) \times Temp11(x,y)$

$$Temp12(x,y) = C_{\theta}^{k}(x,y) \times Temp12(x,y)$$

$$Temp21(x,y) = C_{\theta}^{k}(x,y) \times Temp21(x,y)$$

$$Temp22(x,y) = C_{\theta}^{k}(x,y) \times Temp22(x,y)$$

4. Each of these 4 results are then rectified such that any value less than zero is rectified to zero (this is the $\lfloor . \rfloor$ operation) (1 clock cycle)

5. The final left and right border ownership responses are computed (1 clock cycle):

   $B_{Left}^{k}[\theta] = Temp11 + Temp21$

   $B_{Right}^{k}[\theta] = Temp12 + Temp22$

6. These final results are then stored in two separate BRAM in a parallel fashion (1 clock cycle)

This state machine process is a sequential process for each pixel location. Each result is the left and right border ownership responses/maps for 4 edge orientations (0, $\frac{\pi}{4}$, $\frac{\pi}{2}$, and $\frac{3\pi}{4}$) and each of the 3 pyramid levels, in parallel. Therefore, $4 \times 3 = 12$ left and right border ownership responses (maps) are computed in parallel. This totals to 24 responses and therefore, 24 additional BRAMs for storing the responses. The total time for computing the left and right border ownership (4 edge orientations) for each pyramid level can be seen in Equation 3.32. The total additional block RAM

required can be seen in Equation 3.33.

$$\begin{aligned}&\texttt{Time for Left/Right BO Computation (level)}\\&= N_{rows}(level) \times N_{cols}(level) \times 6\ \texttt{CC}\end{aligned} \tag{3.32}$$

$$\begin{aligned}&\texttt{Additional BRAM (level)}\\&= Precision \times N_{rows}(level) \times N_{cols}(level)\\&\times 4\ \texttt{(Four Oriented Edges)} \times 2\ \texttt{(Left and Right BO)}\\&= 8\ \texttt{bits} \times N_{rows}(level) \times N_{cols}(level) \times 8\end{aligned} \tag{3.33}$$

Table 3.12 summarized the total time and additional block RAM required for computing the left and right border ownership responses for all 4 orientations and 3 pyramid levels.

3.4.4.7 P7: Grouping Responses

The grouping stage consists of various steps (See Fig. 3.12). The first is transmitting the left and right border ownership responses for all pyramid levels (×3) and all orientations (×4) to the PC/MATLAB. The max border ownership orientation and direction (left or right) border ownership masks are computed (See Section 2.5.2). This results in a mask of 1s and 0s for each of the 24 maps (4 orientations, left and right-side BO (×2), and 3 pyramid levels).

These binary masks are computed on PC/MATLAB using max operators and the

Table 3.12: Summary of Time and BRAM Required for All 4 Left BO and 4 Right BO Responses for Each of the 3 Pyramid Levels (×24) (100 MHz Clock)

Model Specification	Value
Total Time (Level 1, 8 Responses)	56,448 CC (564.48μs)
Total Time (Level 2, 8 Responses)	28,800 CC (288.00μs)
Total Time (Level 3, 8 Responses)	14,784 CC (147.84μs)
Additional BRAM (Level 1, 8 Responses)	75,264 Bytes
Additional BRAM (Level 2, 8 Responses)	38,400 Bytes
Additional BRAM (Level 3, 8 Responses)	19,712 Bytes
Total Time (All Levels, 24 Responses)	56,448 CC (564.48μs)
Additional BRAM (All Levels, 24 Responses)	133.376 KB

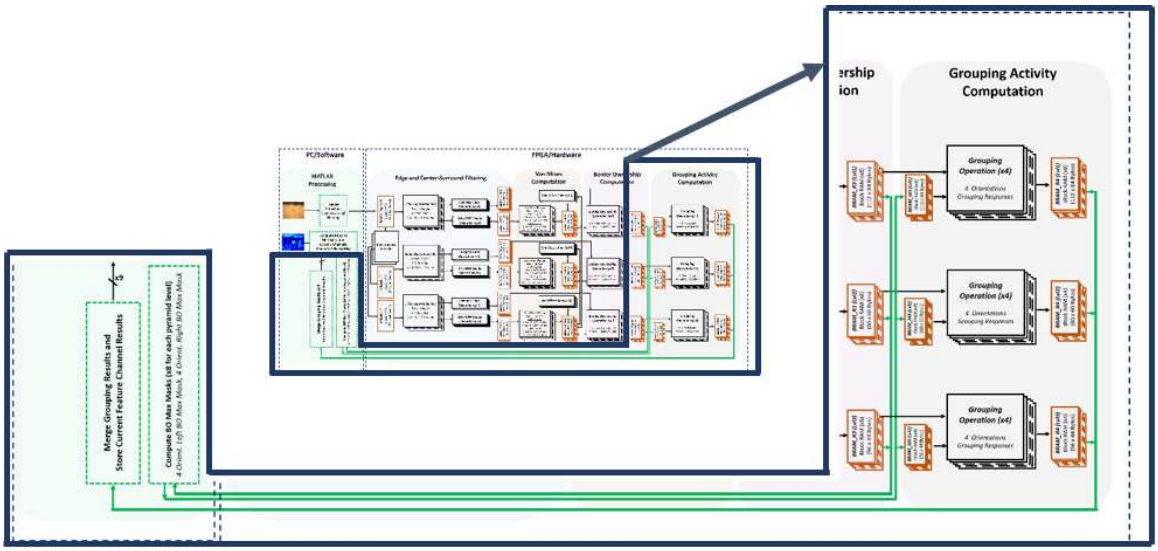


Figure 3.12: Grouping Activity Computation Component of FPGA Model

left and right border ownership responses. The masks are then transmitted via USB to the FPGA for the final grouping computation. The total time for transmitting to MATLAB, computing the masks, and transmitting back to the FPGA is $\sim 34.1ms$.

Once these two binary masks are received by the FPGA ($BOMaskLeft^k_\theta(x,y)$ and $BOMaskRight^k_\theta(x,y)$) for all pyramid levels (k) and orientations (θ), the grouping responses for left border ownership and right border ownership are computed independently using Equations 3.34 and 3.35, respectively.

$$\begin{aligned} GrpLeft^k_\theta(x,y) = BOMaskLeft^k_\theta \otimes B^k_{Left}[\theta] * v_\theta \\ -w_p \times BOMaskLeft^k_\theta \otimes B^k_{Right}[\theta] * v_\theta \end{aligned} \tag{3.34}$$

$$\begin{aligned} GrpRight^k_\theta(x,y) = BOMaskRight^k_\theta \otimes B^k_{Right}[\theta] * v_{\theta+\pi} \\ -w_p \times BOMaskRight^k_\theta \otimes B^k_{Left}[\theta] * v_{\theta+\pi} \end{aligned} \tag{3.35}$$

The parameter w_p is the weight of the inhibitory connection to the opposing side border ownership. The operator $\otimes$ is an element-wise operation. The final grouping step involves summing the left and right grouping responses as seen in Equation 3.36.

$$GrpSum^k_\theta(x,y) = GrpLeft^k_\theta(x,y) + GrpRight^k_\theta(x,y) \tag{3.36}$$

This results in 4 grouping responses for each orientation for each pyramid level ($\times 12$). A state machine is used for computing the grouping responses above in parallel (all 4 orientations and 3 pyramid levels) via the following steps:

1. Retrieve the address of the current pixel on which to perform the grouping response (1 clock cycle)

2. Load current 5×5 image patch at current pixel location (125 clock cycles)

3. Compute 4 weighted sums as seen below using v_θ and $v_{\theta+\pi}$ kernels (75 clock cycles):

   $WS1_{Left} = B^k_{Left}[\theta] * v_\theta$

   $WS2_{Left} = B^k_{Left}[\theta] * v_{\theta+\pi}$

   $WS1_{Right} = B^k_{Right}[\theta] * v_\theta$

   $WS2_{Right} = B^k_{Right}[\theta] * v_{\theta+\pi}$

4. Load current binary mask values for both $BOMaskLeft^k_\theta$ and $BOMaskRight^k_\theta$ (1 clock cycle)

5. Compute the following two responses seen Equations 3.34 and 3.35 (2 clock cycles)

6. Perform a rectification such that if any of these results are less than zero, rectify to the value zero (1 clock cycle)

7. Compute summation to get final grouping response as seen in Equation 3.36 (1 clock cycle)

8. Store result in block RAM (1 clock cycle)

Given these required steps for each pixel location of the current map, the total time required for extracting the grouping response for a given pyramid level and orientation can be computed as seen in Equation 3.37. Note that all 4 orientations for each of the 3 pyramid level responses are computed in parallel state machines. The total additional BRAM required for storing the grouping responses for each pyramid level can be seen in Equation 3.38. This step requires $\times 4$ additional BRAM for each pyramid level ($\times 3$).

$$\begin{aligned}
&\texttt{Total Time for Grouping Response (level)} \\
&= N_{rows}(level) \times N_{cols}(level) \times (1\ \texttt{CC} + 125\ \texttt{CC} + 75\ \texttt{CC} + 1\ \texttt{CC} + 2\ \texttt{CC} \\
&\qquad +1\ \texttt{CC} + 1\ \texttt{CC} + 1\ \texttt{CC}) \\
&= N_{rows}(level) \times N_{cols}(level) \times 207\ \texttt{CC}
\end{aligned} \tag{3.37}$$

$$\begin{aligned}
&\texttt{Additional BRAM (level)} \\
&= Precision \times N_{rows}(level) \times N_{cols}(level) \times 4\ \texttt{(Four Orientation Responses)} \\
&= 8\ \texttt{bits} \times N_{rows}(level) \times N_{cols}(level) \times 4
\end{aligned} \tag{3.38}$$

These 12 results are transmitted to MATLAB and summed across all 4 orientations for each pyramid level. This results in a single grouping response for each of the 3 pyramid levels for the current feature channel. The time for transmitting and computing this final summation is $\sim 4.2ms$. A summary of the time and additional

BRAM required for the grouping process on FPGA can be seen in Table 3.13.

Table 3.13: Summary of Time and BRAM Required for Grouping Responses for Each of the 3 Pyramid Levels (100 MHz Clock)

Model Specification	**Value**
Total Time (Level 1, 4 Responses)	1,947,456 CC ($19.47ms$)
Total Time (Level 2, 4 Responses)	993,600 CC ($9.93ms$)
Total Time (Level 3, 4 Responses)	510,048 CC ($5.10ms$)
Additional BRAM (Level 1, 4 Responses)	37,632 Bytes
Additional BRAM (Level 2, 4 Responses)	19,200 Bytes
Additional BRAM (Level 3, 4 Responses)	9,856 Bytes
Total Time	$6.1ms$ + 1,947,456 CC ($0 - 19.47\mu s$) + $4.2ms \approx 10.3 - 30ms$
Additional BRAM	66.688 KB

3.4.5 Sequential Grouping Computation within Each Channel

The grouping activity for each pyramid level is computed for each of the 9 feature channels sequentially. Therefore, the time required for computing the grouping activity for a single feature channel must be multiplied by 9 to compute the total time

for computing the grouping activity in all feature channels (See Equation 3.39.

$$\begin{aligned}\texttt{Total Time for Grouping Activity Within Single Feature Channel}\\ &= T_{FeatureExtraction} + T_{GroupingComputation} \times 9 \\ &\approx 2.1ms + \sim 51ms \times 9 \\ &\approx 461.1ms\end{aligned} \tag{3.39}$$

3.4.6 Final Normalization and Saliency Map Generation

The final normalization stage and saliency map generation is identical to that seen in the previous Chapter in Section 2.5.4 and also occurs in MATLAB. This final Itti et al. [18] normalization and merging of pyramid levels to generate the final saliency map takes $\sim 20ms$. Henceforth, the total time for computing saliency on a single frame is $\sim 481ms$ (frame rate of $\sim 2Hz$).

Ideally, computing grouping activity within multiple feature channels in parallel is ideal. This is possible with a smaller resolution input image of 60×80 as three feature channels are computed in parallel. We can achieve 2.5× speedup with this resolution.

3.5 Results and Discussion

3.5.1 Resources

The resources used for this complete model for both the version for 84×112 and 60×80 resolution version of the model can be seen in Table 3.14. The limiting resources were the available DSP slices and block RAM. Instantiating another module for computing the grouping activity of another feature channel in parallel at 84×112 resolution would utilize too many resources and would not fit on this FPGA. However, for the 60×80, three parallel feature channels is sufficient for fitting on this FPGA.

Table 3.14: Resources Used by OK XEM7350-160T FPGA for DPOVS-ST Model

Resource	**Available**	**Used** (84×112)	**Used** (60×80) (**$\times 3$ Channels**)
Slice Registers	202,800	17,764 (8%)	33,911 (16%)
Slice LUTs	101,400	20,166 (19%)	36,313 (35%)
Block RAM (RAMB36E1)	325	139 (42%)	195 (60%)
Block RAM (RAMB18E1)	650	180 (27%)	179 (27%)
DSP Slices (DSP48E1)	600	339 (56%)	594 (99%)

3.5.2 Speed

At a resolution of 60×80, the FPGA-based DPOVS-ST model has a runtime of $\sim 21.3ms$ per channel. The MATLAB model has a runtime of $\sim 125.6ms$ per channel. This is a speed-up of about $6\times$ with using the FPGA implementation of the model. For a fair comparison, we made the same modifications made to the FPGA model, also to the MATLAB model. The advantage of the FPGA model is its ability to process multiple feature channels in parallel. There are 9 total feature channels. In our current implementation, we run 3 channels in parallel. This allows for a frame rate of $15.64Hz$ compared to the MATLAB model of $0.89Hz$. This is a $\sim 15\times$ speed-up. Furthermore, if we implemented all 9 feature channels in parallel on an FPGA that had a sufficient amount of resources, we can achieve $50\times$ speed-up. These results can be seen in Fig. 3.13.

3.5.3 Accuracy

To determine the accuracy of the model, we use the ImgSal v1.0 dataset by Jian Li from the National University of Defense Technology in Changsha, P.R. China [119]. This dataset consisted of 60 static images with attached fixation data. We used the KLD metric to compute how well the MATLAB and FPGA model's saliency map predict human eye fixations. Ideally, the average KLD for the MATLAB model and FPGA model should be relatively the same. As you can see in the table in Fig. 3.13,

Specification	MATLAB	FPGA	
Runtime [per channel]	~125.6 ms	~21.3 ms	**6x speed-up**
BRAM [per channel]	-	66.69 KB	
# Feature Channels	9	9	
Resolution	60 x 80 (or 84 x 112)	60 x 80 (or 84 x 112)	
Pyramid Levels	3	3	
Kernel Size	5 x 5	5 x 5	

Specification	MATLAB	FPGA *3 Parallel Channels*	FPGA *9 Parallel Channels*
Framerate	0.89 Hz	**15.64 Hz**	46.94 Hz
Speed-up from MATLAB	-	**17.5x**	52.7x

KLD Divergence (ImgSal v1.0 Dataset)

Matlab Model	FPGA Model	FPGA from MATLAB
1.41	1.50	0.14

Figure 3.13: Comparison of FPGA model to MATLAB model.

the KDL of the MATLAB model is 1.41 and the FPGA model is 1.50. These two values are close as expected. Furthermore, we computed the KL divergence between the distribution of saliency values from the MATLAB model and that of the FPGA model. Ideally, the KLD value should be low since there should be minimal divergence from each other. As expected, we get a low KLD value of 0.14 for this metric. This quantitatively validates our FPGA model's ability to compute an accurate saliency map which predicts human eye fixations.

We also compare the results between the FPGA and MATLAB model visually. Fig. 3.14 shows a side-by-side comparison of the saliency map output for the MATLAB based model (using the same model specifications and input resolution) to the saliency map output of the FPGA-based model for 84×112 visual input resolution. Fig. 3.15 shows the same results for the FPGA-based model for 60×80 visual input resolution.

3.6 Conclusion

In this chapter we demonstrate an FPGA-based implementation of the dynamic visual saliency model, DPOVS-ST for real-time processing. We utilize an Opal Kelly XEM7350-160T Kintex-7 FPGA board with a 100 MHz clock. This model is capable of computing a dynamic visual saliency map on 84×112 resolution visual input at a frame rate of 2.079 Hz. It is capable of computing the same on 60×80 resolution visual input at a frame rate of 5.190 Hz. To our knowledge, this is the first FPGA

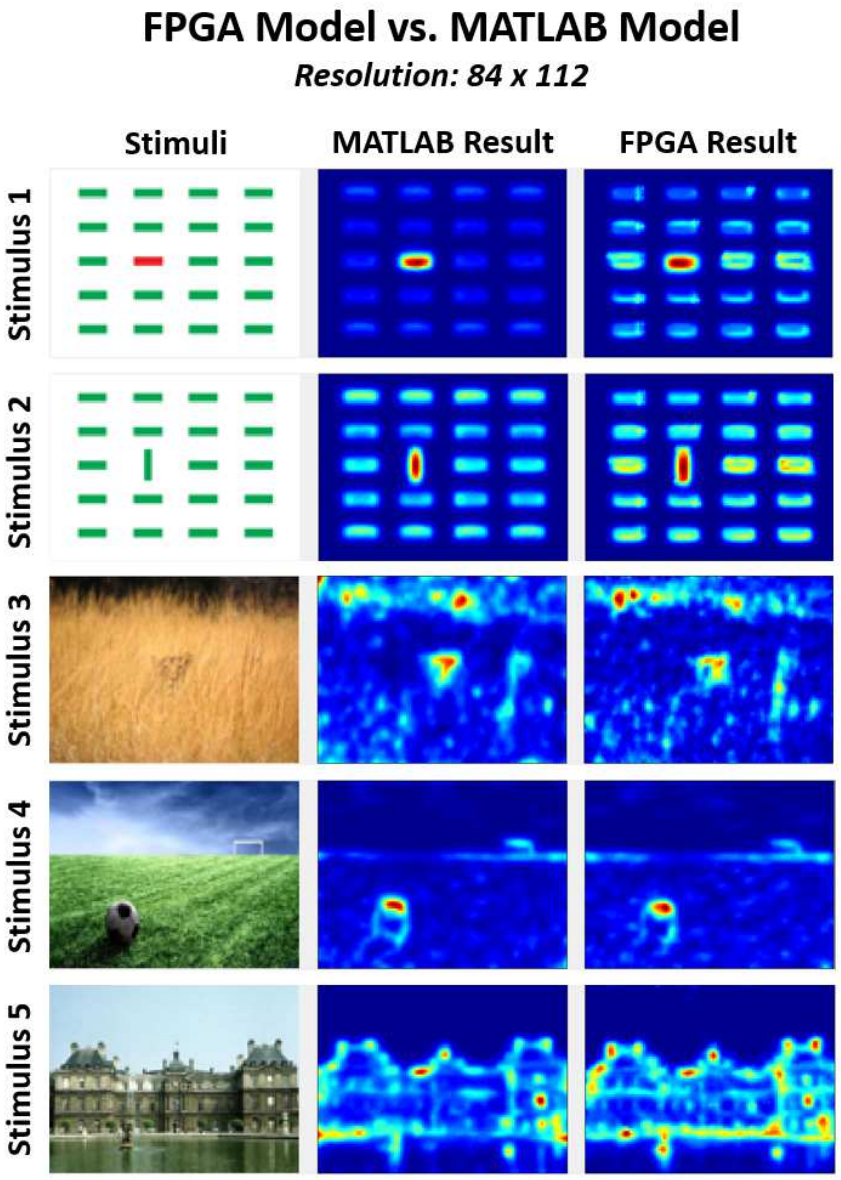


Figure 3.14: Results from saliency map computed (on 5 arbitrary images) from FPGA model compared to that from MATLAB model for 84×112 *resolution visual input.*

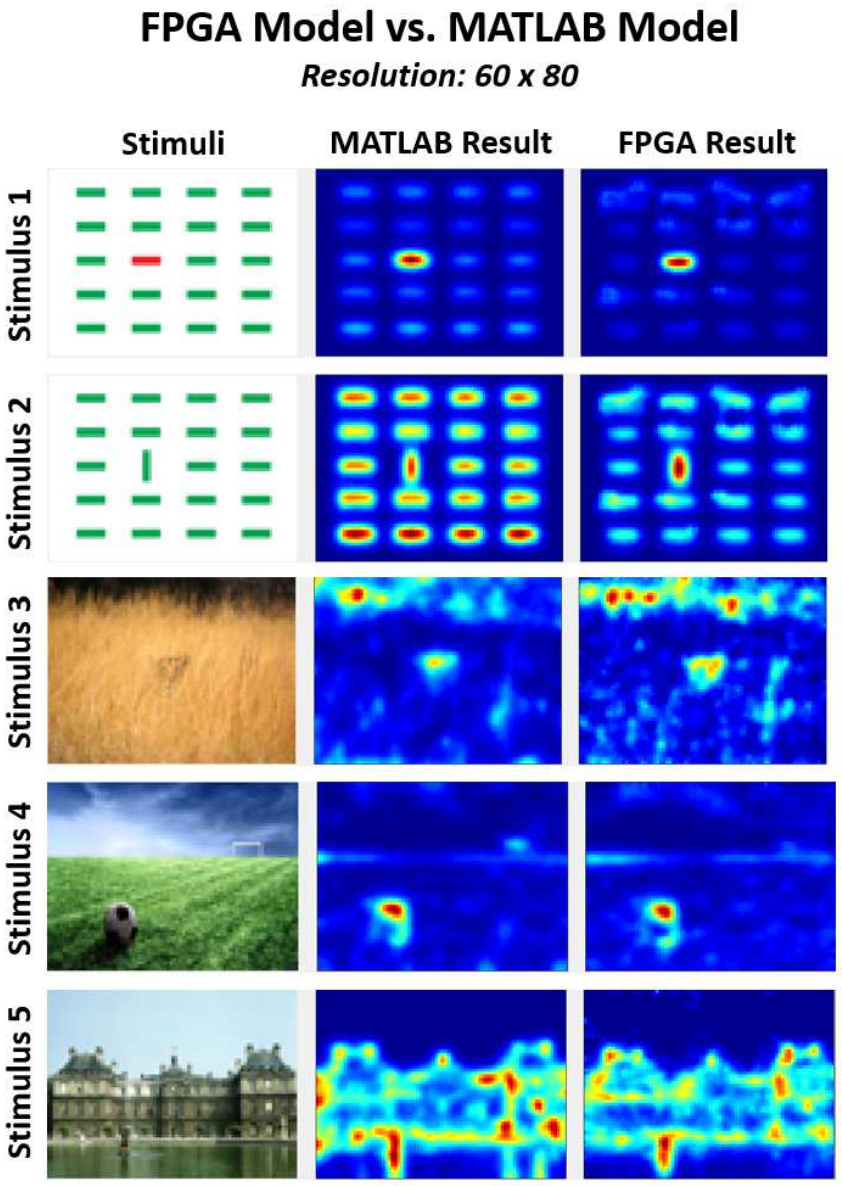


Figure 3.15: Results from saliency map computed (on 5 arbitrary images) from FPGA model compared to that from MATLAB model for 60×80 *resolution visual input.*

implementation of an object-based (and proto-object-based) visual saliency model. Many hardware-based models ([114], [117], [112], [116]) compute saliency based on features and do not require the computation of pre-attentive objects. This additional computation of pre-attentive objects is a computational burden and not easily implemented in hardware. With this novel hardware implementation, we demonstrate the ability to compute an object-based visual saliency model in hardware and pave the path for other similar models to be implemented in hardware for real-time processing. Considering it has been shown that we perceive the whole of an object prior to its features ($[1, 60, 91, 92]$), such a model is necessary for accurate computation of saliency. This hardware-based model presents many interesting applications which can utilize dynamic saliency computation at its front-end (to "filter out" irrelevant information) considering its small-size, light-weight, low-power platform.

Chapter 4

FPGA Model: Integrate and Fire Array Transceiver

4.1 Overview

4.1.1 FPGA-based IFAT

In the preceding chapters, a novel dynamic visual saliency model was discussed along with its hardware implementation. The model is biologically-plausible and depicts a vision task that can be coupled to any visual processor as a front-end data triaging solution. In this chapter we discuss a generic system capable of performing event-based image processing tasks, specifically visual pre-processing tasks, which can be utilized by the visual saliency model (previously discussed) or other visual system.

CHAPTER 4. FPGA IMPLEMENTATION OF THE INTEGRATE AND FIRE ARRAY TRANSCEIVER

This system is an FPGA implementation of the Integrate-and-Fire Array Transceiver (IFAT). In this chapter we will discuss two approaches to modeling the IFAT on FPGA (FPGA-IFAT). The first is one using stochastic computational elements [23]. This exploits the stochastic properties of neurons in the brain. Events are generated using stochastic computational elements for performing multiplications using simple digital elements (i.e. AND gates). As with any event-based system, information is represented in spikes/events over time in stochastic event streams. We demonstrate a real-time simultaneous dewarping and filtering task using this FPGA implementation of the IFAT. Finally, we generalize the IFAT system to directly mimic a VLSI implementation of the IFAT [25]. This FPGA-based system accepts address events as inputs and outputs address events. This facilitates integration with any other event-based system. We demonstrate this by integrating with an event-based camera called the ATIS (Asynchronous Time-based Image Sensor). The ATIS serves as the front-end to the FPGA-based IFAT system. We demonstrate a real-time simultaneous dewarping and spreading operation using this complete end-to-end event-based system. The FPGA implementation of the IFAT serves as an ideal system for real-time validation of systems which would further run on a VLSI-based IFAT system. However, it is also in a ready-state to be used directly with systems which may take advantage of its relatively low-power, minimal data throughput, and small-size specs as an FPGA, event-based visual processor.

4.1.2 Dewarping Application

Advancements in the field of neuromorphic engineering has allowed for the development of efficient systems which are constrained by area and power. More specifically, designing around these constraints is especially important for micro aerial vehicles (MAVs). The use of autonomous MAVs pose many advantages in applications including military, surveillance, and data acquisition when large systems are detrimental. Henceforth, there is a great amount of interest in the advancement of technologies that have fascimiles to natural fliers such as insects and birds [120, 121]. In order for MAVs to navigate autonomously, they must be able to perform processing on visual information in real-time as it is consistently in motion. The raw visual information is warped (translated, rotated, etc.) in relation to the camera motion, and henceforth, it must be dewarped into a single 2D space. It is important to perform image dewarping on the initial visual stimulus in order to perform accurate further processing on the visual information (i.e. object detection, object recognition, learning, tracking, etc.). In this work, image dewarping is performed using event-based processing and communication [14]. This is similar to the communication protocol and processing exhibited by neurons in our brain. This neuromorphic approach to dewarping allows us to perform such computation under low-power, small area, and light-weight constraints.

4.2 Related Work

Over the past few decades, with the development of FPGAs, there as been emerging research involving the implementation of spiking neural networks on an FPGA platform for real-time processing. FPGAs are ideal in that they allow for fast prototyping, reusability and reconfigurability, and small-size. Typically, the neural networks implemented on FPGA are targeted for learning applications. Many models implement the feed-forward, classification stage on FPGA while the training occurs in software [122–124]. The trained weights are loaded onto the FPGA for classification via the FPGA-based neural network. Eldredge et al. [125] implemented their RRANN (Run-Time Reconfiguration Artificial Neural Network) model on FPGA. In their work, they successfully implemented a backpropagation algorithm on a Xilinx XC3090 FPGA. Backpropagation is necessary for training the neural network, allowing for not only a feed-forward network, but also backpropagation and update stages for learning. Similarly, Glackin et al. implemented both the training stages and feed-forward stages for classification on FPGA [126].

In this work, we are focused on a generalized approach to neural network implementation on FPGA. We are interested in implementing a generic array of neurons with synaptic connections stored in memory (look-up-table), allowing reconfiguration of the network depending on the application. Such a generic FPGA implementation of a generic, reconfigurable spiking neural network was implemented on FPGA by others. Cassidy et al., implemented an array of 32 neurons capable of STDP (Spike

Time-Dependent Plasticity) [127]. Finally, Bade et al. implemented a spiking neural network of 15 neurons and 75 synapses using stochastic computational elements for performing weighted sums on a Xilinx 4003PG120-5 FPGA [128].

In this work we focus on implementing the IFAT on an FPGA-based platform. We design for optimizing speed, resources, and reconfigurability to allow for any type of connectivity within the spike-based neural network using an AER communication protocol. We specifically demonstrate how we can configure this system for performing a simultaneous dewarping and filtering task.

4.3 Integrate-and-Fire Array Transceiver (IFAT)

The IFAT was briefly introduced in Chapter 1. A high-level diagram of the IFAT system can be seen in Fig. 4.1. The IFAT system is a bio-inspired asynchronous event-based system which allows for reconfigurability for performing various event-based tasks [14,77]. The system sends and receives address events using time-division multiplexing. Address events (AEs) are asynchronously transmitted over a single bus. In the same manner, AEs are received asynchronously over a single bus. Incoming AEs are representative of presynaptic action potentials, which are first sent to a look-up-table (LUT) containing the destination addresses and corresponding weights for the incoming AE. This is representative of the synaptic connections between neurons.

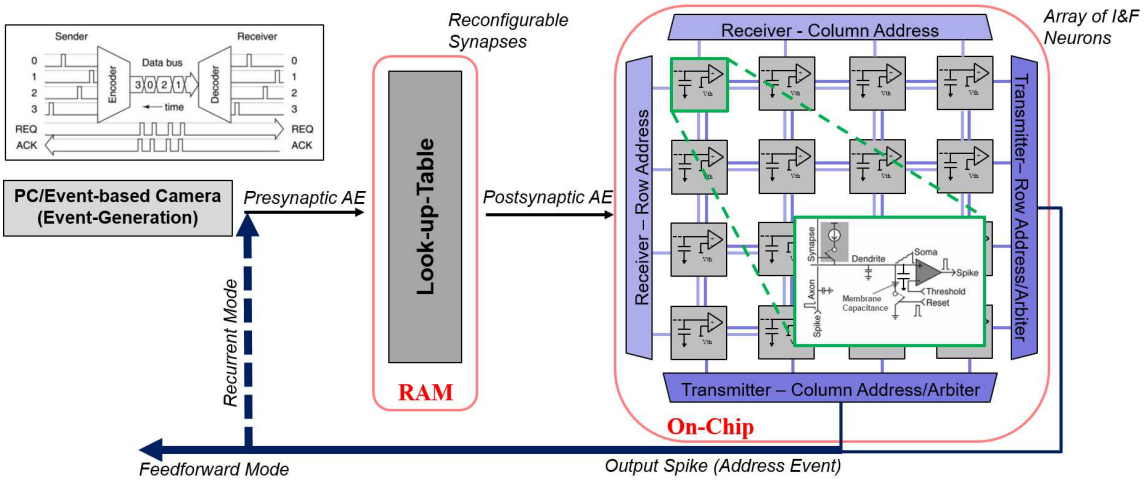


Figure 4.1: Block Diagram of the Integrate-and-Fire Array Transceiver (IFAT). Presynaptic events go through a LUT which holds postsynaptic connections with corresponding weights. These postsynaptic events are sent to the array of integrate-and-fire neurons. As neurons reach their threshold and generate events, the events are outputted via the AER-based transmitter.

Events are then sent to the appropriate destination addresses within the IFAT array (with their appropriate weights) representing postsynaptic action potentials. The IFAT array consists of an array of neuron circuits which contain a capacitor representing the neuron membrane capacitance. As a neuron receives events, a "packet" of charge proportional to the weight between the presynaptic event and postsynaptic event (designated within the LUT) is integrated on to the membrane capacitance. The connection between two neurons can be either inhibitory or excitatory. Inhibitory connections are represented by negative weights and excitatory connections are represented by positive weights. When the voltage across the capacitor (representing the membrane voltage of the neuron) reaches a globally set threshold, the supporting circuitry within each neuron allows for the neuron to output an AE corresponding to the address of the neuron sending the event. The voltage across the neuron that outputted the spike is immediately reset to a globally set reset voltage (typically zero). Address events (AEs) are received via the receiver and outputted via the transmitter. It is also important to note that there is an unlimited number of synaptic connections using this approach. All of the network connections are established within the LUT. The LUT can be reprogrammed dynamically, "on-the-fly", allowing for interesting applications including real-time dewarping.

4.4 FPGA-based IFAT Model Using Stochastic Computational Elements

Regarding the IFAT, in summary, the IFAT model contains an array of elements modeled as integrate-and-fire (I&F) neurons which integrate charge as they receive spikes ("events") and then emit and transmit spikes when they reach a specified threshold [14]. An address-event protocol is used for communication. Information is encoded in the events similarly to the encoding and decoding of spiking neurons in the brain. The key idea behind the IFAT is that it allows for dynamic reconfigurability of the routing of incoming events. We exploit this capability for performing an image dewarping task. With readily available data pertaining to camera motion, real-time dewarping can be computed by reconfiguring the look-up-table (LUT) to remap pixels to their dewarped location. Considering all pixel intensities are encoded within the probabilistic timing of events, this system is ideal for low-power constraints and robustness to noise.

In this work, we further extend the IFAT system to take advantage of stochastic computation [129]. The spike-based communication scheme used by neurons in our brain is probabilistic by nature. Henceforth, by incorporating stochastic elements in our model, we take one step closer to modeling a system that closes the performance gap between our brain and computer systems. Stochastic computation (SC) allows for encoding numbers in bit streams opposed to the conventional binary represen-

tation of values. Utilizing SC elements is ideal for various reasons. First, the bit streams can easily be integrated into the IFATs event-based protocol. Secondly, SC elements consume low area and low power. Lastly, they are noise-tolerant considering its probabilistic nature. Recently, a number of systems have shown success in performing unconventional computations by using SC elements [122, 130]. In this model, SC elements are integrated in the event generation and communication scheme for representing pixel intensities while simultaneously having the capability of performing multiplications, and in the same manner, weighted sums.

We present a Field Programmable Gate Array (FPGA) implementation of an IFAT-based model coupled with SC elements for performing real-time dewarping on 40×40 pixel video frames. The board is an Opal Kelly 6310-LX150 containing a Spartan-6 FPGA. Using known camera motion and video data from a commercial quadcopter, we demonstrate our systems capability of performing the dewarping task with minimal error when validated against performing the task using conventional dewarping operations computed in MATLAB.

4.4.1 System Architecture

The complete model of the FPGA-based architecture can be seen in Fig. 4.2. While the circuit components could not be directly implemented on the FPGA, the FPGA system's functionality is representative of its VLSI counterparts. In the following sections we will first discuss the system's input. Secondly we will discuss the

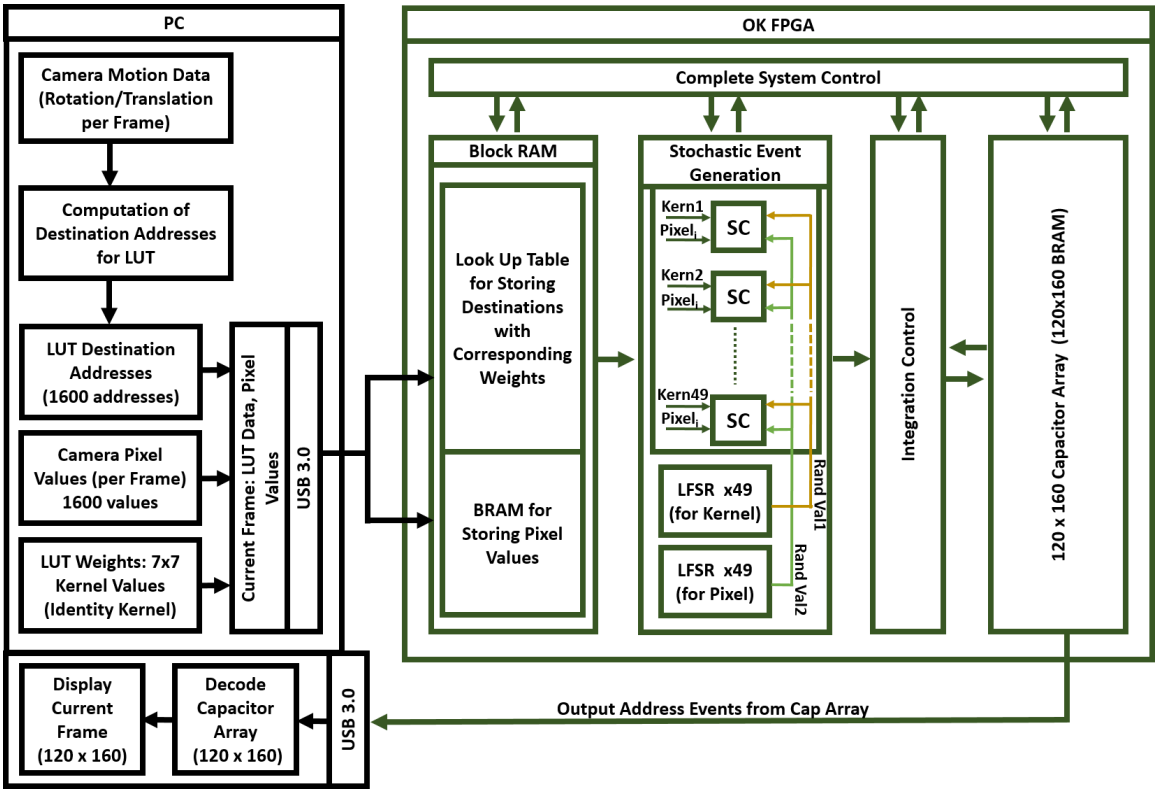


Figure 4.2: Block Diagram of the FPGA implementation of the IFAT integrated with Stochastic Computational Elements. Image from [23].

FPGA architecture in regards to stochastic event generation and IFAT implementation. Finally, we will discuss how we validated the system with respect to an image dewarping task and compare these results to a MATLAB-based system.

4.4.1.1 Input

We process a given video frame by frame. Each frame contains 40×40 grayscale pixel intensity values with attached camera rotation and translation data at each frame. It is important to note that in this work, we assume we can extract camera motion (using an accelerometer fixated to the camera). For this first implementation, a traditional, frame-based camera is used with attached rotation and translation data about a single 2-D coordinate system. This rotation and translation data is computed with respect to an arbitrarily selected reference point (See Fig.‘4.3).

We use N-bit precision where N is a parameter set prior to running the system. With the use of stochastic computation, higher precision allows for better accuracy but slower speed. For each frame, we send three sets of data from the PC to the FPGA via USB 3.0 communication. The first consists of the 40×40 N-bit pixel intensities of the current frame representing values between 0 and $2^N - 1$. The second set consists of the destination addresses associated with each pixel location in the current 40×40 pixel frame. The destination addresses are computed via MATLAB using the camera motion data to dewarp each pixel in the frame. Each location in the frame is remapped to its respective destination address(es) in the 120×160

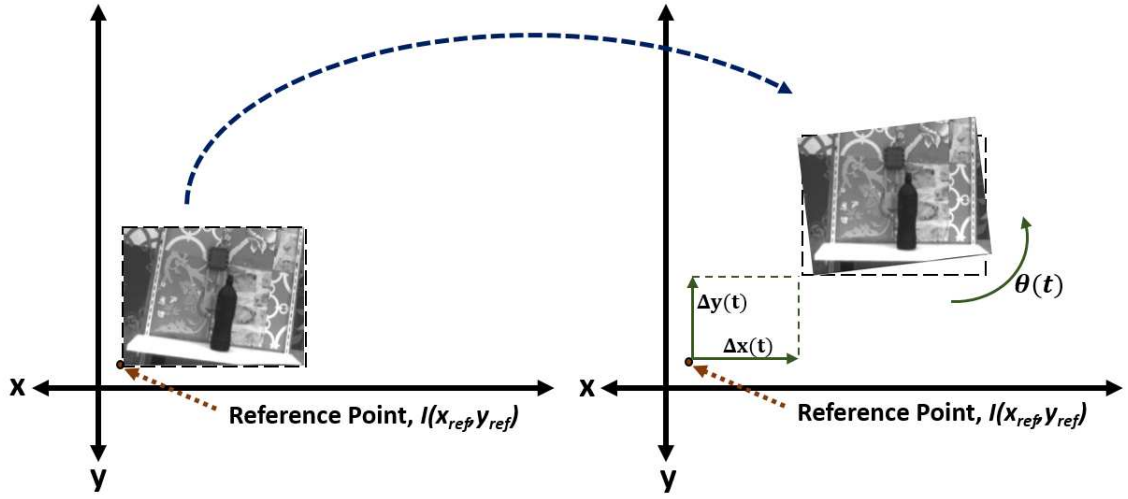


Figure 4.3: Demonstration of dewarping image. The translation (Δx and Δy) along with rotation (θ) with respect to an arbitrarily selected reference point ($I(x_{ref}, y_{ref})$) is obtained via an accelerometer and gyroscope fixated to the camera.

neuron array. Note that the neuron array resolution (120×160) is higher than the input image resolution (40×40). Assuming the camera motion data is synchronized with the camera frame output, we assume that we have instantaneous rotation and translation data attached to each frame output. Assuming we have selected a reference point, ($I(x_{ref}, y_{ref})$) in the neuron array, and we know the instantaneous translation in the $x-$direction and $y-$direction (x_{trans} and y_{trans}, respectively), and we also have obtained instantaneous rotation (θ) with respect to the center of the image/frame (x_{center}, y_{center}), we can then compute the destination address (x_{dest}, y_{dest}) for each pixel (x, y) using a rotation matrix (See Equation 4.1).

$$\begin{bmatrix} x_{dest} \\ y_{dest} \end{bmatrix} = \begin{bmatrix} cos(-\theta) & -sin(-\theta) \\ sin(-\theta) & cos(-\theta) \end{bmatrix} \begin{bmatrix} x - x_{center} \\ y - y_{center} \end{bmatrix} + \begin{bmatrix} x_{center} \\ y_{center} \end{bmatrix} + \begin{bmatrix} x_{trans} \\ y_{trans} \end{bmatrix} + \begin{bmatrix} x_{ref} \\ y_{ref} \end{bmatrix} \quad (4.1)$$

Finally, the synaptic weights associated with the destination address(es) of each pixel location are sent to the FPGA. For performing a filtering task, these weights are determined from a 7×7 kernel set prior to running the system. Henceforth, in this system, each location in the frame will have 49 destination addresses with corresponding weights. This is analogous to a single presynaptic address event having 49 different postsynaptic address events with different synaptic weights. A simplified visual example of this can be seen in Fig. 4.4. The synaptic weights are initially normalized between $[0, 1]$. However, before programming the LUT, the associated synaptic weights are normalized between $[0, 2^N - 1]$, and therefore, is a function of the precision. Note that the precision of the pixel intensity and precision of the weights must be the same.

After computing the destination addresses and corresponding weights, the following data is transmitted to the FPGA:

- Pixel intensities of current frame
- Destination addresses with corresponding synaptic weights

The total amount of BRAM required for storing the pixel intensities of a single frame can be seen in Equation 4.2.

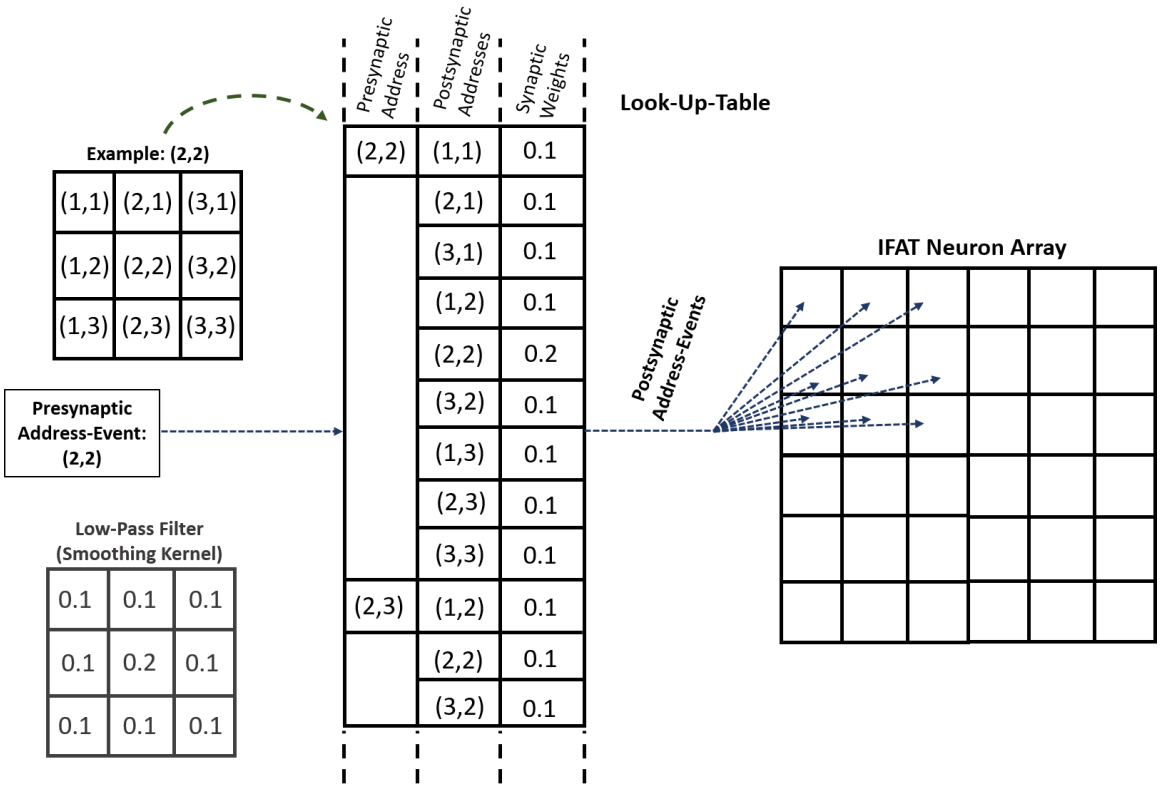


Figure 4.4: Visualization of IFAT flow and how LUT is programmed. For performing a filtering task, the kernel weights are used for determining corresponding weights for synaptic connections within the 3×3 neighborhood. An example using pixel location $(2, 2)$ can be seen here.

$$\texttt{BRAM for Pixels: Number of Bits} = I_{Rows} \times I_{Cols} \times N \tag{4.2}$$

The variables I_{Rows} and I_{Cols} is the number of rows and columns in the image, respectively. The value N is the bit-precision. For example, if we are using 8-bit precision, the total BRAM required for storing the pixels of a single frame is $40 \times 40 \times 8 = 12,800$ bits $= 1.6KB$.

The total amount of BRAM required for storing the destination addresses and weights (assuming a maximum of 49 synaptic connections per neuron) can be seen in Equation 4.3.

$$\texttt{BRAM for LUT: Number of Bits} = (\lceil log_2(IFAT_{Rows} \times IFAT_{Cols} \rceil + N) \times 49 \times I_{Row} \times I_{Cols} \tag{4.3}$$

The value $\lceil log_2(IFAT_{Rows} \times I_{Cols} \rceil$ is the number of bits required for representing all addresses in the $IFAT_{Rows} \times IFAT_{Cols}$ neuron array. For example, given 8-bit precision, the total BRAM required for the LUT is $(log_2(120 \times 160) + 8) \times 49 \times 40 \times 40 = 1,803,200$ bits $= 225.4KB$.

The total required BRAM for the input for various precision, N, can be seen in Table 4.1. Fig. 4.5 shows the total required BRAM for the input as a function of precision.

Table 4.1: BRAM Required for Various Precision for a 40 × 40 Resolution and 49 Synaptic Connections per Neuron

Parameter	$N = 4$	$N = 6$	$N = 8$	$N = 10$	$N = 12$
Storing Pixels BRAM	0.8 KB	1.2 KB	1.6 KB	2.0 KB	2.4 KB
Storing LUT BRAM	186.2 KB	205.8 KB	225.4 KB	245.0 KB	264.6 KB
Input: Total BRAM	187.0 KB	207.0 KB	227.0 KB	247.0 KB	267.0 KB

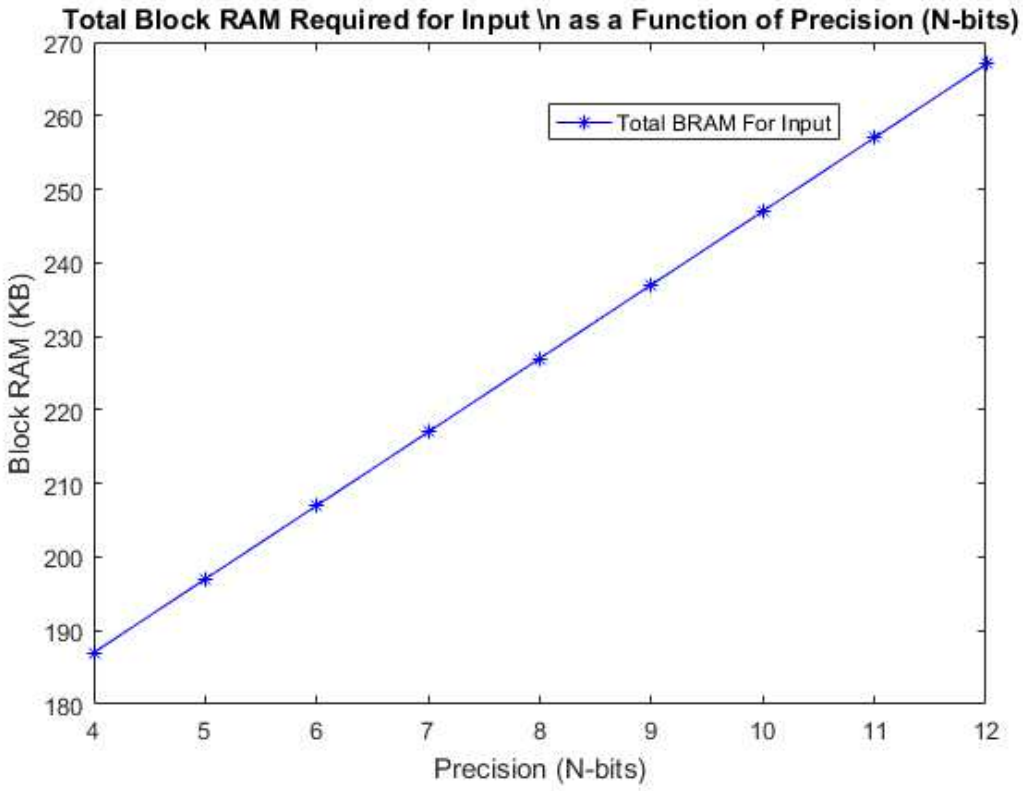


Figure 4.5: Total required BRAM for the input of the FPGA-based IFAT with SC as a function of N-bit precision.

4.4.1.2 Stochastic Computational Event Generation

When all data for the current frame is received by the FPGA, the processing on the FPGA is initiated. At the start of processing, the LUT contains the destination addresses associated with each incoming pixel address within the 40×40 frame. The weights for each destination address are also stored in the LUT. The next step involves stochastic computation for event generation. Fig. 4.6 shows a single stochastic computational block). There are 49 SC blocks operating in parallel corresponding to the 49 synaptic connections per neuron. This allows for 49 postsynaptic event streams for a single presynaptic address event. In stochastic computation, digital values are represented as probabilities within event streams. An event stream contains X_0 number of $0s$ and $X_1 = 2^N - X_0$ number of $1s$ given N-bit precision. To generate event streams, a comparator is used to compare the N-bit value to be represented within the bit stream against a generated pseudorandom N-bit value. At each clock cycle, the N-bit value is compared against a new N-bit pseudorandom number from a uniform distribution using an N-bit linear-feedback shift register (LFSR) of maximal length. If the value is greater the generated pseudorandom number (at the current clock cycle), than a 1 (signaling an event) is generated. Otherwise, a 0 is generated. The result is an event stream consisting of 2^N bits where $\frac{X^1}{2^N}$ is the probability representing a specific value (e.g. a bit stream of 1 1 1 0 1 0 1 1 1 0 1 1 0 1 1 1 corresponds to $0.75 \times 2^4 = 12$). In this system we utilize two comparators for generating two different bit streams for each of the 49 SC blocks. One bit stream represents the current

pixel value ($Pixel_i$), and the second represents the synaptic weight ($Kern_j$ where j = 1,2,3,...,49). Considering the requirement of 2^N length event streams, there is a trade-off between precision and time when using stochastic computation (See Fig. 4.7.

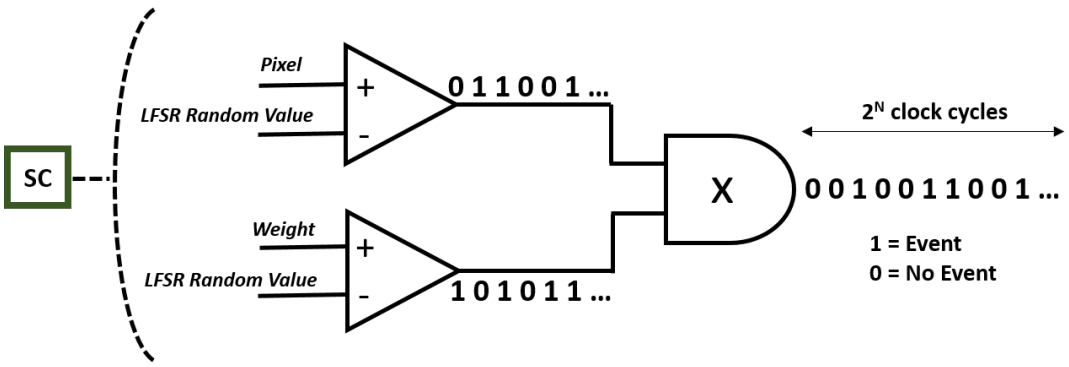


Figure 4.6: Stochastic computational element used for event generation within the FPGA IFAT implementation. A comparator is used for generating stochastic event streams for both the pixel value and weight/kernel value. An LFSR is used for generating pseudorandom values. An AND gate is used for performing the multiplication between the pixel and weight/kernel.

4.4.1.3 Neuron Array and Charge Accumulation

Each SC block performs a multiplication between the $Pixel_i$ event stream and $Kern_j$ event stream (Fig. 4.6). This generates the final postsynaptic event stream integrated onto the appropriate destination capacitor element within the 120 × 160 array. By using stochastic computation, we can perform this multiplication using a single AND gate [129]. The input into the AND gate are the two event streams representing $Pixel_i$ and $Kern_j$. The output event stream from the AND gate repre-

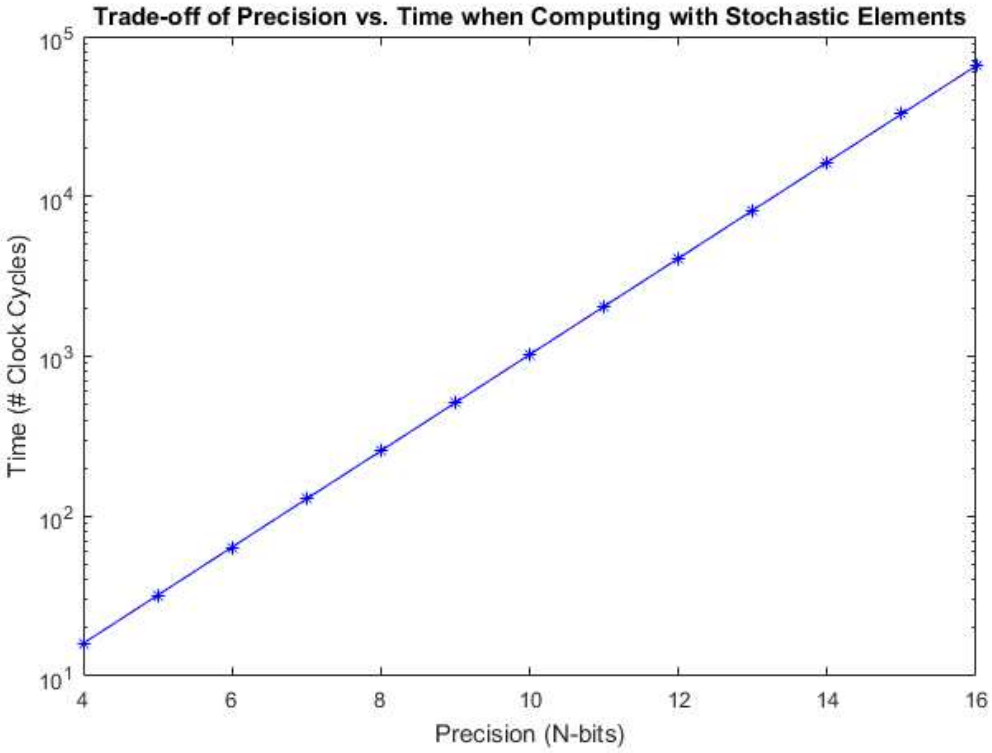


Figure 4.7: Trade-off of precision vs. time when using stochastic computational elements for generating events and performing accurate multiplication.

sents $Pixel_i \times Kern_j$. When performing multiplication using an AND gate as the SC element, two properties must be satisfied. The first is that the input event streams must be uncorrelated. Secondly, the event streams must be generated from independent random number generators. Henceforth, for generating the pseudorandom N-bit values for each SC block, two independent LFSRs are used (as seen in Fig. 4.2). One LFSR is used for generating the pseudorandom values for the event stream representing the kernel value/weight ($Kern_j$). The second LSFR is used for generating the pseudorandom value for the event stream representing the pixel value ($Pixel_i$). This method of multiplication is advantageous in that it requires minimal hardware area to perform an N-bit multiplication.

The postsynaptic event stream accumulation is remapped/dewarped to its appropriate location in the 120×160 neuron array. In our FPGA emulation, this results in an accumulation of 1s over the event stream's integration time (2^N clock cycles). This summation is then accumulated onto the corresponding destination cell (membrane potential) in the 120×160 capacitor array and stored in its appropriate position in BRAM. The control integration block in Fig. 4.2 controls the accumulation of events onto its appropriate destination in the 120×160 BRAM capacitor array. The accumulation of events is representative of the accumulation of charge onto capacitor elements (soma) within the VLSI-based IFAT system. The neurons' current membrane potential is stored in BRAM until it is accessed again by postsynaptic events.

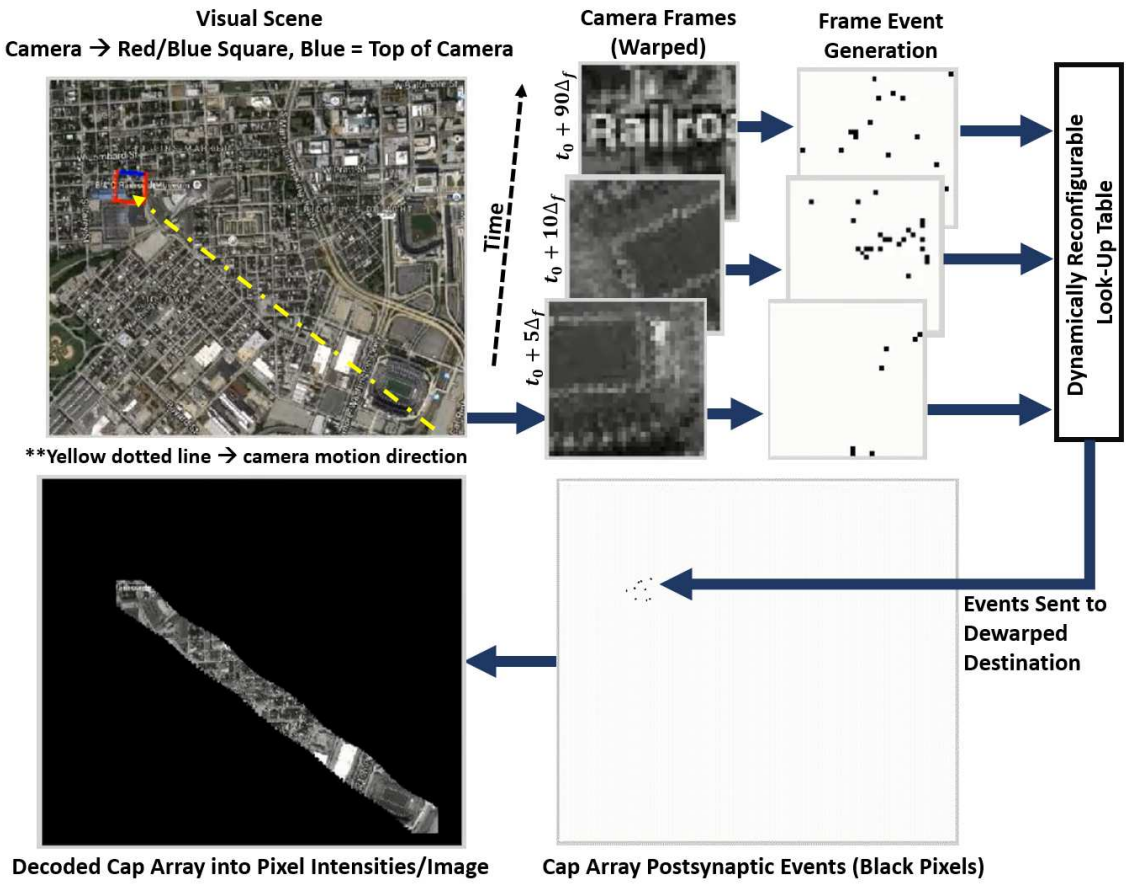


Figure 4.8: Screenshot of the working model using simulated moving camera data from a Baltimore City aerial view. This shows event generation at various frames starting a $t_0 + 5\Delta_f$, $t_0 + 10\Delta_f$, $t_0 + 90\Delta_f$, *where* t_0 *is the time at the start of processing and* Δ_f *is the time to process a single frame. The LUT is reconfigured dynamically at each frame for rerouting the incoming events. Image from [23].*

4.4.2 Results and Discussion

4.4.2.1 Resources

The resources utilized for this complete model can be seen in Table 4.2. Although we generated an implementation for 4-bit, 6-bit, 8-bit, 10-bit, and 12-bit precision, in this table, we show the lowest and highest precision results ($N = 4$ and $N = 12$). It is also important to note that no DSP slices were required by using stochastic computation for performing multiplications.

Table 4.2: Resources Used by OK XEM6310-LX150 FPGA for FPGA-based IFAT Model with Stochastic Computational Elements (for $N = 4$ and $N = 12$)

Resource	**Available**	**Used ($N = 4$)**	**Used ($N = 12$)**
Slice Registers	184,304	1,742 (1%)	3,002 (1%)
Slice LUTs	92,152	1,927 (2%)	2,872 (3%)
Block RAM (RAMB16BWER)	16	268 (5%)	17 (6%)
Block RAM (RAMB8BWER)	5	536 (1%)	5 (1%)
DSP Slices (DSP48A1)	0	180 (0%)	0 (0%)

4.4.2.2 Speed

To determine the speed of the system, we first consider only processing that occurs on the FPGA. Assuming the current frame and LUT has already been stored, the

timing can be split into the following steps using a finite state-machine:

1. Read Pixel Value at Current Location in Frame
2. Read Destination Addresses and Weights
3. Stochastic Event Generation
4. Read Current Neuron Potential at Destination Addresses
5. Add Event Accumulation Results to Current Neuron Potential
6. Write Back to Memory
7. Repeat Previous Steps for Each Location in Frame

The first step involves reading the pixel value at the current location requiring 1 clock cycle. Reading the destination address and weights requires latching the address, wait for the value to be available, and then latching the result (for each of the 49 synaptic connections). This results in 3 clock cycles per synaptic connection ($\times 49$). The following step involves the stochastic computation for event generation. This step requires 2^N clock cycles. The next step involves reading the current membrane potential value at the 49 different destination addresses, updating these values, and then writing the updated result back to memory. This process requires 3 clock cycles per synaptic connection ($\times 49$). This entire process is then repeated for each location in the frame (40×40). Equation 4.4 depicts the number of clock cycles required for computing a single frame as a function of bit precision, N.

$$\texttt{Number of Clock Cycles/Frame} = (1 + 3 \times 49 + 2^N + 3 \times 49) \times I_{Rows} \times I_{Cols}$$
$$= (2^N + 295) \times 1600 \tag{4.4}$$

Fig. 4.9 is a plot depicting the frame rate as a function of bit precision, N, and running on a 100 MHz clock.

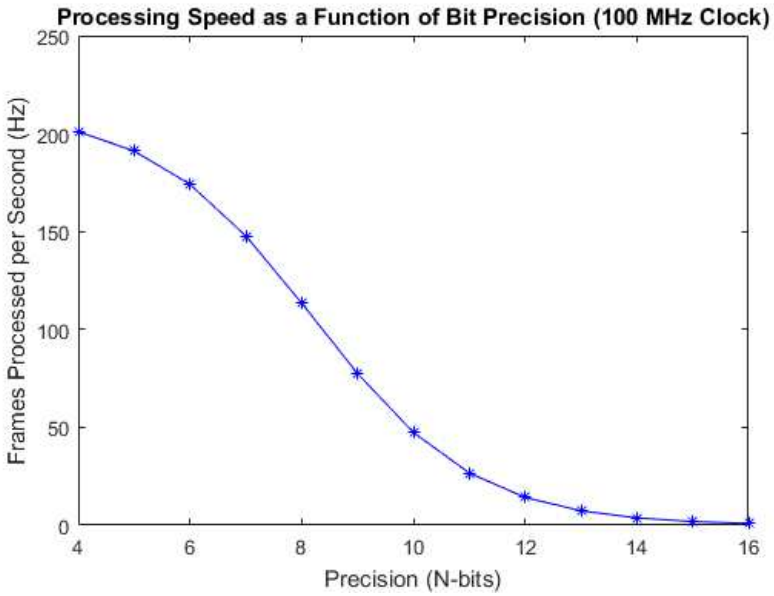


Figure 4.9: Speed of the IFAT FPGA processing as a function of bit precision, N, given a 100 MHz clock.

4.4.2.3 Block RAM Utilization

The amount of block RAM required for this FPGA-based IFAT implementation using stochastic elements is a function of the precision (N), image/frame resolution

($I_{Rows} \times I_{Cols}$), and IFAT neuron array size ($IFAT_{Rows} \times IFAT_{Cols}$). The Equation 4.5 depicts the amount of BRAM required as a function of the bit precision, N. Fig. 4.10 visualizes this equation.

$$\begin{aligned} \texttt{Total BRAM} &= \texttt{BRAM for Input} + \texttt{BRAM for IFAT Array} \\ \texttt{Total BRAM} &= \big((\lceil log_2(IFAT_{Rows} \times IFAT_{Cols}\rceil + N) \times 49 \times I_{Row} \times I_{Cols}\big) + \\ &\quad \big(IFAT_{Rows} \times IFAT_{Cols} \times N\big) \end{aligned} \tag{4.5}$$

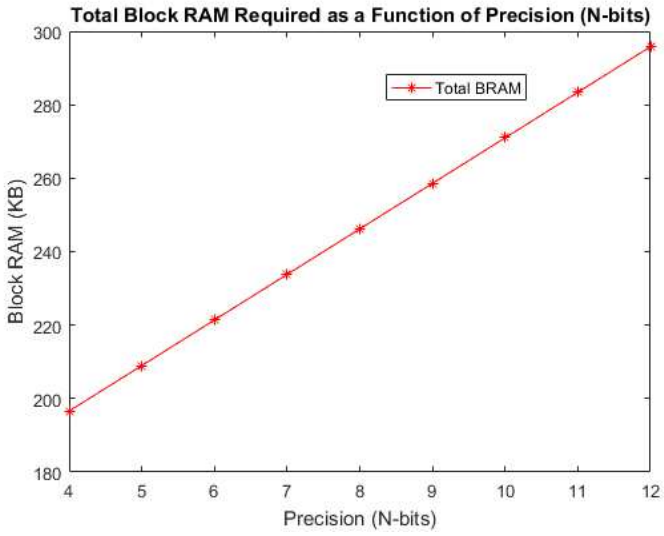


Figure 4.10: BRAM required for the IFAT FPGA model as a function of bit precision, N.

4.4.2.4 Validation

4.4.2.4.1 Experimental Setup

We validate this systems ability to perform a dewarping task by comparing this stochastic IFAT-based systems output to that computed via MATLAB. A commercial AscTec Pelican quadcopter with an attached camera is used for obtaining the video data with corresponding camera rotation and translational motion data (gyroscope and accelerometer data). The frame rate of the camera was 35 Hz. The camera was attached on top of the quadcopter facing directly ahead parallel to the ground such that the 2-D video frames of the scene were perpendicular to the ground. Pointing directly up corresponds to the z-axis, left-to-right corresponds to the x-axis, and forward-to-backward corresponds to the y-axis. We limited the movement of the quadcopter such that only translational movement along the x-axis and z-axis and rotation about the y-axis was allowed. This is due to the limitation of our current system to consider change in scale, and henceforth, distance along the y-axis remained constant. The motion data contained quaternion rotation and translation data which we use to determine the rotation and translation matrix for dewarping each pixel address to its appropriate destination address. For the MATLAB implementation, we dewarped each input pixel intensity values at each frame to its corresponding dewarped location in the 120×160 complete image scene. This was considered ground truth. For our system, we compute the destination address for each pixel location at each frame and use the destination addresses as input to the system. A

total of 2280 input frames of video was obtained of which we selected five different sets of 40 continuous frames with corresponding camera motion data for validating our model.

4.4.2.4.2 Results

The working system is visualized in Fig. 4.8. To numerically compare our systems output to ground truth, we first run the system over all 40 frames of a given set. The pixel intensities in each results are scaled to the range 0 to 255. We then compute the absolute difference of each pixel intensity in the resulting 120×160 image from the MATLAB-computed output. The average across all absolute differences is used to obtain the average difference in pixel intensity. This average is divided by the maximum difference of 255 (representing maximum error) to give the percentage of error. We run the system and compute the error for 4-, 6-, 8-, 10-, and 12-bit precision. Table 4.3 shows the resulting average error over all 5 sets of data for different N-bit precision. Furthermore, we show the speed and stochastic computation integration time (in clock cycles) for the various precisions (N).

Our results show we can accurately perform the dewarping task with minimal error. At 4-bit precision, the systems output has an average error of 6.61% which is equivalent to an average error of approximately 17 in pixel intensity difference. However, the processing speed is faster at 14.13 ms per frame. With a trade-off in

Table 4.3: Results and Validation of IFAT-based System's Dewarping Task Output Against MATLAB Dewarping Task Output

Precision	**Avg. Error**	**Speed (Frames/Sec)**	**SC Integration**
4-bit	6.61%	70.77 Hz	16 clock cycles
6-bit	5.27%	65.96 Hz	64 clock cycles
8-bit	4.86%	52.03 Hz	256 clock cycles
10-bit	1.77%	28.30 Hz	1024 clock cycles
12-bit	1.61%	10.02 Hz	4096 clock cycles

speed, we can achieve better accuracy at 12-bit precision with an average error of 1.61% which is an approximate error of 4 in pixel intensity difference. At 12-bit precision, each frame is processed in 99.81 ms. The precision at which the system should run at is completely dependent on the application. Ideally, for devices such as MAV, faster processing speed is ideal. However, depending on the processing tasks to be performed on the dewarped output, high accuracy may also be necessary. On the contrary, post-processing on the dewarped output may be robust enough to handle more error, which will allow for lower-bit precision and faster speeds. This low-power, event-based system emulation demonstrates the ability to produce sufficient dewarped output of a moving camera even at lower-bit precisions.

4.5 FPGA-based IFAT Model Generalized

In the previous model described, we presented a system in which a traditional frame-based camera was used as input to the system and stochastic computational elements were used for generating stochastic event streams based on the pixel intensities and corresponding synaptic weights. In the following model, we present an end-to-end stochastic, event-based neuromorphic system capable of performing a dewarping task while simultaneously performing a smoothing task via a 3×3 filter. For aerial vehicles, such as micro-aerial vehicles (MAVs) and other unmanned aerial vehicles (UAVs), to accurately perform tasks such as object detection, recognition, and tracking, the dewarping task is critical. As the aerial vehicle navigates throughout its environment, the camera also moves, and therefore, the raw camera output is warped in a manner that follows the camera motion. Therefore, output must be initially dewarped so that post-processing can be performed on an accurately registered 2-D mosaic of the environment that the camera has captured. Kim et al. [131] demonstrated a similar dewarping task, however, they use event output from a different event-based image sensor [15] and use a Bayesian approach for estimating camera motion. In our work, we reduce computational complexity and increase speed by assuming camera motion is readily available from a device such as an Inertial Measurement Unit (IMU).

One of the key novelties in this work is the integration of the ATIS camera previously discussed (event-based camera). In our prior work, an ATIS camera was not

used. Instead, a standard digital camera with raw digital output consisting of an array of 8-bit RGB pixel values was used. Stochastic event streams were generated using stochastic computational (SC) elements. In this current work, the ATIS camera output is naturally in the stochastic, event-based representation necessary for the IFAT implementation. The use of the ATIS removes the need for the SC elements to generate an event-stream, and instead, allows us to have a complete end-to-end event-based system for performing the dewarping and filtering task. We then demonstrate how the reconfigurability component of the IFAT system can allow us to perform these tasks in real-time for applications such as aerial vehicles.

4.5.1 System Architecture

In the subsequent sections we will discuss each component of the system from end-to-end. The ATIS camera and its event-based, stochastic output will be discussed. The IFAT system and FPGA implementation of this system will then be discussed. Finally, we will explain how we use this system to perform simultaneous dewarping and filtering for aerial vehicles. The full system bock diagram can be seen in Fig. 4.13.

4.5.1.1 ATIS Event-based Output

In the previous work discussed in Section 4.4 [23], we used SC elements for generating stochastic event streams from 8-bit pixel values. However, for an end-to-end event-based system, the camera itself should be inherently event-based. For this sys-

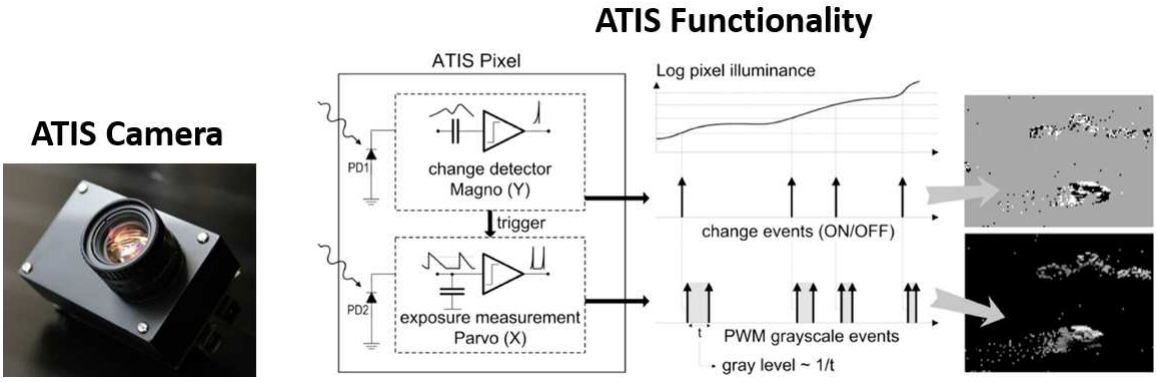


Figure 4.11: ATIS camera and visualization of its functionality. The ATIS detects log intensity change and outputs a single address event (TD event) depicting there was a change and whether it was a positive or negative (ON or OFF) change. This event is immediately followed by two additional address events (EM events) in which the time (t) between events is inversely proportional to the intensity at the pixel which exhibited a change. Image from [24].

tem [25] we integrate the ATIS camera [16] and generalize the FPGA-based IFAT system. The ATIS, a silicon retina image sensor, is different than traditional frame-based cameras. Each pixel within the ATIS consists of a change detector circuit which senses relative log intensity changes in illumination similarly to the Dynamic Vision Sensor (DVS) [15]. It outputs an event when it senses a change. A second and third event is outputted representing the time to a low and high threshold-crossing of a photocurrent integration at the pixel which exhibited the log intensity change (See Fig. 4.11. The ATIS utilizes an address event representation (AER) communication protocol such that the output events of all pixels are time multiplexed and share the same bus. Each event is the address of the pixel that most recently triggered an event.

The first event (change event) is further tagged with an additional bit representing the direction of change (ON/OFF event). Time is an inherent component of the system. The pixel intensity at each address is then decoded temporally as it is represented within the ISI (Inter-spike-interval) at which the pixel outputs an address event after it has exhibited a sufficient relative change in intensity. Such an event-based approach to camera output poses many advantages. These include low data throughput, no temporal resolution limitation in regards to frame-rate, high dynamic range (on the order of 143dB), low power consumption and low area [16].

4.5.1.2 Stochastic Component

In regards, to the stochastic component of our system, we utilize the natural stochastic properties of the ATIS camera. The events generated output event stream generated is a function of various factors which are inherently noisy. This noise induces the stochastic component of the system. Such stochasticity is ideal because it better mimics the communication scheme of neurons in the brain and is useful for systems which exploit stochasticity in their computation methods. Aside from randomness in illumination exhibited externally (i.e. within illumination) stochasticity comes from three primary sources:

1. As with any CMOS (Complementary Metal-Oxide Semiconductor) technology, there is stochasticity that arises from the fabrication process. There consists mismatch and random telegraph noise within CMOS technology. Random tele-

graph noise is caused by the random capture and release of charge carriers in the bulk of transistors [132]. This may cause current spikes resulting in random output of events at pixels that do not actually exhibit relative intensity change.

2. The second source of stochasticity is within the arbitration circuitry of the AER architecture. The AER circuitry contains an arbitration circuit that randomly selects a single pixel address when multiple pixels have triggered an event at the same time [16].

3. Finally, the ATIS camera contains a mode that allows for a continuous random background refresh in which photo measurement in each pixel is triggered by the randomly distributed parasitic leakage in the change detector reset transistors [16].

A combination of these stochastic components of the ATIS allows for generation of event streams which are a function of relative change in illumination, as well as the inherent stochasticity of the system. To quantify the stochasticity we computed the distribution of random events across the pixel array. To do this, we placed a blank sheet of paper directly in front of the ATIS lens for achieving constant illumination across the pixel array. We then summed the total number of events within various time windows. The total number of random events within a time windows of 1 ms was counted. This process was repeated for 500 independent time windows of the same duration. We then determine the distribution of total number of events across

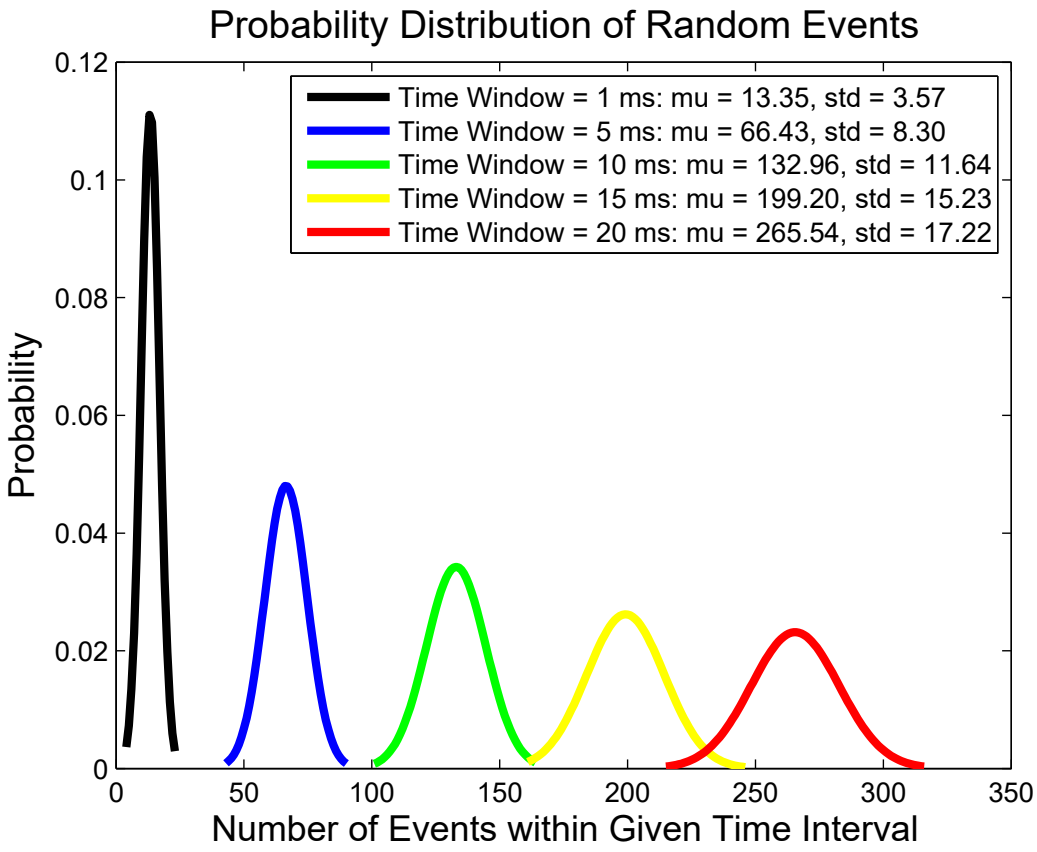


Figure 4.12: Probability distribution of random events occurring within various time windows.

all 500 independently sampled time windows. This same experiment was also conducted for time intervals of 5 ms, 10 ms, 15 ms, and 20 ms. Fig. 4.12 shows the fitted probability distribution for the five different time windows. As seen in this plot, the distribution of random events follows a normal distribution with a specific mean and standard deviation. As the time window increases, the mean and standard deviation increases, as expected. However, the normal distribution of random events is maintained regardless of the duration of the time window. These results confirm that there are in fact random events occurring and that they follow a normal probability distribution. The random events follow a normal distribution regardless of the time window in which the events are sampled. Showing the stochasticity component of the system allows our system to further mimic the low-power properties and sparse communication scheme of biological neurons. Our system is then able to take advantage of models which require a stochastic, event-based input.

4.5.1.3 FPGA Model

The complete system can be seen in Fig. 4.13. The input of the FPGA-based IFAT system is the output address event stream from the ATIS camera. For our system, we only consider the two events outputted from a pixel after a change in intensity has been detected. The ATIS camera is coupled to a second, independent FPGA containing a module for receiving events and outputting them to the PC. Considering the x- and y-dimension of the ATIS is 304 $\times$ 240, respectively, each event consists

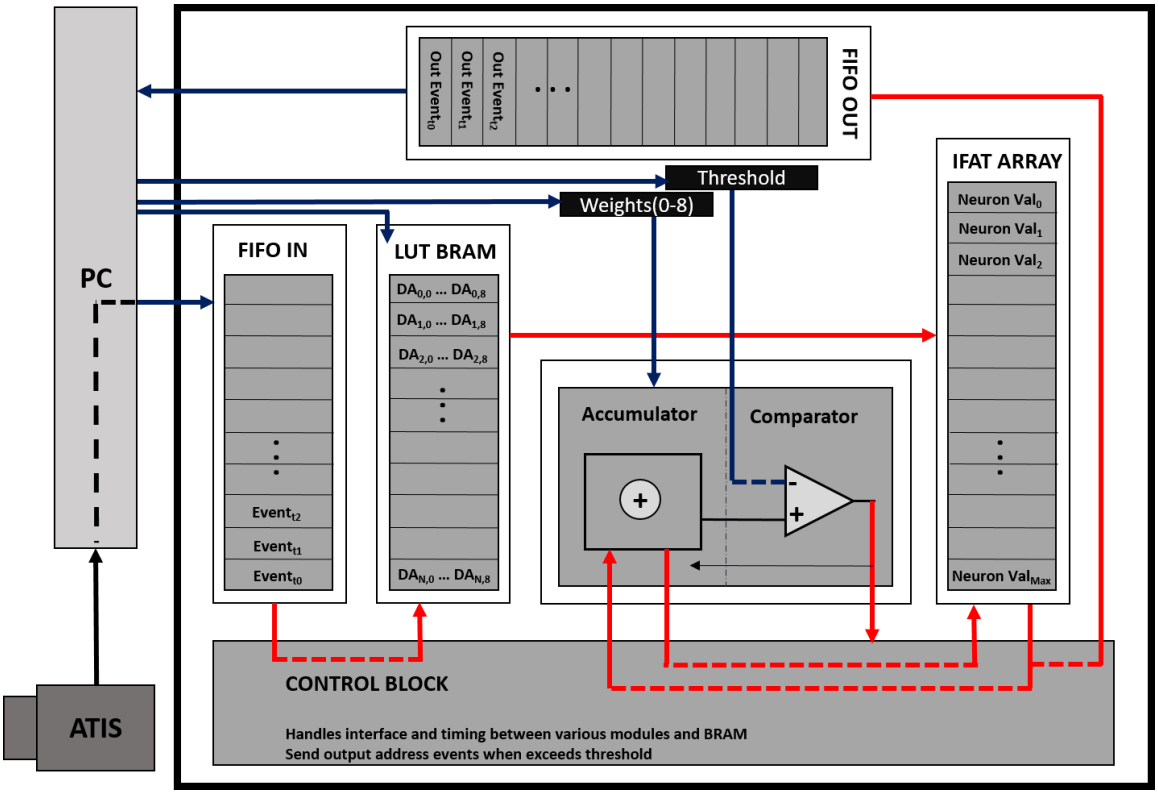


Figure 4.13: Block diagram of the complete system operating on stochastic, event streams outputted from the ATIS camera. Image from [25].

of the an x- and y-address representing the address of the pixel that outputted the event. It also consists of a timestamp (derived from an FPGA clock) and a single bit signifying whether it was the first or second event. It is important to note, however, that for our IFAT system, the only useful information is the address of the event since time is inherent within the system. When the system receives an event, the address is extracted and latched onto the address select of the LUT. This LUT is implemented as a Block RAM on the FPGA. The values stored in this LUT are the destination addresses in the neuron array for each incoming address event. In our case, considering the system performs a 3×3 filtering task, each value in the LUT Block RAM will contain nine destination addresses with corresponding weights (stored in a separate register array). The weights are simply a value corresponding to the number of events that should be sent to each destination address. The neuron array is implemented as a Block RAM consisting of 175,000 8-bit values. This corresponds to a neuron array with x- and y-dimensions of 500×350 with accumulation up to a value of 255. A module consisting of a digital accumulator coupled with a clocked comparator is used for reading the value at the destination address, adding the weight, and determining if the new value is above a globally-set threshold (with a max value of 255). If it is greater than the threshold, that neuron outputs an event to the output FIFO and the neurons value in Block RAM is reset to a global-reset value (typically 0). This output event consists of its address in the neuron array, and again, time is inherent within the system. This output FIFO continuously outputs events to the PC as it

receives a sufficient number of output events from the neuron array.

Finally, the output event stream seen by the PC can be used for post-processing. In our case, we continuously decode the pixel intensity at each address simply by counting number of events received at each address over some interval of time.

4.5.1.4 Dewarping

A key component of this system is its capability of performing a dewarping task in real-time, a task that is essential for aerial vehicles which must do further processing on its camera output. To reduce computational complexity and increase speed of the system, we assume that camera motion data can be readily obtained in real-time. Henceforth, we use a commercial IMU, which outputs both translational and rotational motion. We then use this data to determine how the camera has been warped and apply a rotation matrix to dewarp the camera output (via programming the LUT of the system appropriately). For this work, we constrained camera motion to only translational motion about a 2-D coordinate plane and rotation parallel to the same plane and assume scene structure is not dependent on translation. Henceforth, the equations for determining the dewarped output of a camera that has rotated by some angle, θ, and translated in the $x-$ and $y-$ direction by x_{trans} and y_{trans}, respectively, are depicted in Equations 4.6 and 4.7. These equations are based on the rotation matrix seen in Equation 4.1.

$$x_{new} = x_{init}cos(-\theta) - y_{init}sin(-\theta) + x_{trans} \tag{4.6}$$

$$y_{new} = y_{init}cos(-\theta) + x_{init}sin(-\theta) + y_{trans} \tag{4.7}$$

where (x_{init},y_{init}) is the initial incoming address and (x_{new},y_{new}) is the final destination address. To perform this dewarping real-time as the camera moves, we use the IMU output to continuously compute the dewarped location of each pixel within MATLAB. We then reprogram the LUT on the FPGA by sending the computed new destination addresses to the LUT Block RAM. Considering our constraint on camera motion, this was sufficient for demonstrating the systems ability to perform the dewarping task. However, future work must consider rotation and translation about more than two axes.

4.5.1.5 Filtering

The filtering, as previously discussed is related to the weights of the connections (synapses) to the destination addresses of incoming address events. For our system, a maximum kernel size of 3×3 can be used. The weights are programmed via MATLAB, and furthermore, can also be updated in real-time if necessary. Each weight corresponds to the value which should be added onto the current neuron value stored in the neuron array Block RAM. For this work, we demonstrate the workings of the filtering by applying a simple smoothing operator such that when an address event is received, a single event is also sent to the 3×3 neighborhood of the destination neuron. This smoothing operator occurs simultaneously with the dewarping by simply applying the weights at the neighborhood of the dewarped location. Henceforth,

this demonstrates how the LUT can be exploited to perform real-time, simultaneous dewarping and filtering tasks (especially for aerial vehicles). Fig. 4.3 visualizes this filtering operation via the IFAT.

4.5.2 Results and Discussion

4.5.2.1 Resources

The resources utilized for this generalized FPGA-based IFAT implementation can be seen in Table 4.4. It should be noted that no DSP blocks were required as all processing is event-based and computation occurs over time.

Table 4.4: Resources Used by OK XEM7350-160T FPGA for the Generalized FPGA-based IFAT Model

Resource	**Available**	**Used**
Slice Registers	202,800	1,790 (1%)
Slice LUTs	101,400	1,970 (1%)
Block RAM (RAMB36E1)	325	96 (29%)
Block RAM (RAMB18E1)	650	2 (1%)
DSP Slices (DSP48E1)	600	0 (0%)

4.5.2.2 Speed

The speed of the system is not represented as framerate considering the input is not in frames but in events. When we refer to speed, we refer to the time from receiving the spike on the FPGA to outputting a spike (if an output spike is triggered). This represents the worst-case delay for processing an incoming event. To compute this delay, we first assume address events exist within the input FIFO and the LUT has been programmed with the appropriate synaptic connections. We can then consider the time it takes to perform the following tasks:

1. Read the next address event.
2. Load the corresponding destination addresses and weights from the LUT.
3. For each destination address, load current "membrane potential" from IFAT Array BRAM.
4. Accumulate "charge" onto each neuron.
5. Check new potential against threshold.
6. Output event (to output FIFO) for each neuron that generates a spike.

The time required for reading the next address event is 1 clock cycle. The time required for loading the corresponding destination addresses and weight is also 1 clock cycle. Loading the current "membrane potential" from the IFAT Array BRAM for each of the destination addresses requires 2 clock cycles per destination (maximum

of 9 destination addresses). Accumulation of charge onto each of these neurons can occur in parallel in 1 clock cycle. The check against the threshold can also occur in a single clock cycle. The worst-case delay occurs when each of these 9 neurons generate an event, in which case each needs to be written to the output FIFO. To write a single address event to the output FIFO requires 2 clock cycles for latching the address to the input of the FIFO and a second clock cycle for writing to the FIFO. The total number of clock cycles required for processing a single event (worst-case) can be seen in Equation 4.8.

$$\begin{aligned}\texttt{Maximum Delay} &= 1+1+2\times\texttt{[Num Dest Neurons]}+1+1+2\times\texttt{[Num Dest Neurons]}\\ &= 40\texttt{Clock Cycles}\end{aligned} \tag{4.8}$$

Therefore, the maximum event rate determined by the FPGA IFAT processing given a 100 MHz clock is seen in Equation 4.9.

$$\begin{aligned}\texttt{Max Event Rate} &= \frac{1}{40\ \texttt{Clock Cycles}\times\frac{1}{100e6}}\\ &= 2.5\ \texttt{Mevents/s}\end{aligned} \tag{4.9}$$

However, the maximum output rate of the ATIS is $\sim 1\mu s$. Therefore, this limits the processing to 1.0 MHz.

4.5.2.3 Block RAM Utilization

The total BRAM required for this FPGA IFAT model is a summation of the input FIFO, LUT, IFAT array, and output FIFO. The total block RAM utilized can be seen in Table 4.5.

Table 4.5: BRAM Used by OK XEM7350-160T FPGA for the Generalized FPGA-based IFAT Model

Model Component	**BRAM Required**
Input FIFO	8.192 KB
Look-Up-Table	164.160 KB
IFAT Array	175 KB
Output FIFO	65.536 KB
Total BRAM	412.888 KB

4.5.2.4 Validation

To demonstrate and validate the functionality of the system, we implemented the FPGA-based IFAT on an Opal Kelly XEM7350-160T consisting of a Kintex-7 FPGA. The system runs on the FPGA's $100MHz$ clock and uses the USB 3.0 FrontPanel interface for communication to and from the PC. The ATIS camera outputs events via an independent FPGA (Opal Kelly 6010-LX150) to the PC.

The purpose of this work is to demonstrate the possibility of attaching this system

to an aerial vehicle and perform the necessary dewarping task as the camera exhibits motion when fixed to the aerial vehicle. To emulate the camera motion, we physically attached the IMU device and ATIS camera. We then pointed the camera at a pair of shoes and rotated the camera around the axes through the center of the lens. The rotation was 45 degrees counter clockwise. First, the ATIS output and IMU data was collected and synchronized via MATLAB. We then used the synchronized data as input to our system. The ATIS output is the stochastic address-event stream that is sent from PC to the FPGA-based IFAT system. The IMU output was used to dynamically reconfigure the LUT for dewarping the ATIS output address events to their appropriate locations in the neuron array. Finally, as previously noted, the weights used corresponded to sending a single event not only to an incoming event's destination address, but also to the 3×3 neighborhood around this destination. This is analgous to a smoothing operator, which is typically used for removing noise in the image at the cost of decreased sharpness of the image. The output event stream represented the dewarped and filtered events which could then be decoded into pixel intensities. Fig. 4.14 shows the visual output of the dewarped and filtered events from the ATIS and IMU data captured.

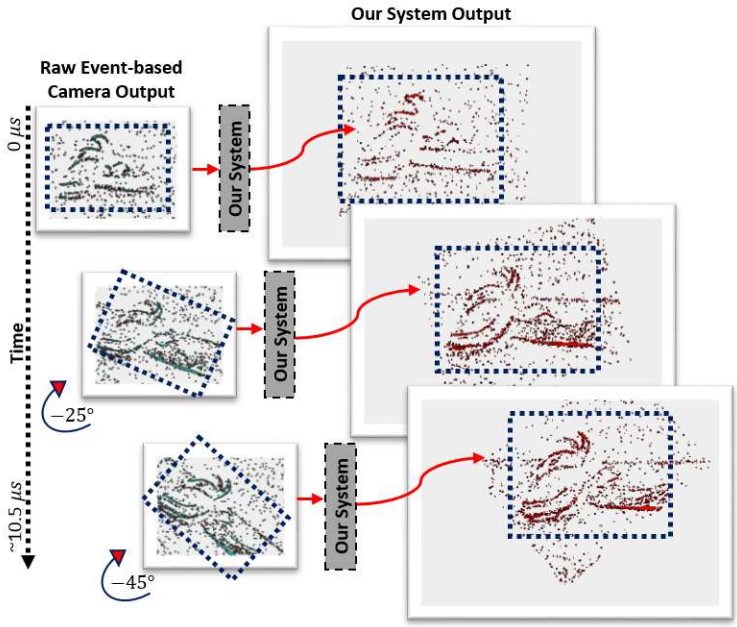


Figure 4.14: Visual demonstration of events being dewarped and filtered through our system at three different points in time.

4.6 Conclusion

We have demonstrated a real-time event-based system emulation using stochastic computational elements for performing a dewarping task implemented on an Opal Kelly FPGA. This system exploits the dynamic reconfigurability of the LUT of the IFAT for remapping pixel locations to its appropriate location on a 120×160 image in real-time. Although using stochastic computation requires a trade-off between speed and accuracy, our results show sufficient accuracy even below 8-bit precision. However, better accuracy can be achieved at higher precision depending on the requirements of the tasks being performed post-dewarping. In future work we seek to extend this model to incorporate scaling, implementation in VLSI technology, and further, interface with a system which can provide real-time camera motion data on an MAV. This work was published and presented at Midwest Symposium for Circuits And Systems (MWSCAS) in 2015 in Fort Collins, Colorado.

We have also demonstrated a generalized, complete end-to-end event-based, neuromorphic system capable of performing a simultaneous dewarping and filtering task on event-based input. We have demonstrated integration with an event-based camera. The system takes us one step closer to a real-time, low-power, low-data throughput system that does processing on events over time opposed to the traditional binary radix representation approach. Such a system is ideal for MAVS and other UAVs which may need to perform image processing tasks in real-time while consuming low power and are limited in regards to data transfer rate. In the future, we seek to

extend this system to a VLSI-technology based implementation of the IFAT system. Furthermore, we seek to experiment with other methods for computing dewarped destination addresses for handling camera rotation and translation across more than two axes. This work was published and presented at International Symposium for Circuits And Systems (ISCAS) in 2016 in Montreal, Canada.

Chapter 5

VLSI System: Mihalas-Niebur Neuron Array Transceiver

5.1 Overview

In the prior chapter, we discussed implementation of a neural array transceiver (IFAT) on an FPGA platform. In this chapter we discuss a novel neural array transceiver implemented in VLSI technology. Implementation in VLSI technology is ideal in regards to the low-power, light-weight, and small-size specifications that we seek. The human brain is by far the most computationally complex, efficient, and robust computing system operating under such low-power and small-size constraints. It utilizes over 100 billion neurons and 100 trillion synapses in achieving these specifications. Within the field of neuromorphic engineering, we seek to de-

sign systems (typically in VLSI technology) which mimic the physical characteristics, functionality, and communication scheme of these neurons. In doing so, this allows for our systems to reach these desirable properties. More obvious applications of these systems involve direct modeling and simulation of neural activity within biological systems. More recently, however, there has been much interest in the design of neural networks for object recognition, classification, and similar visual tasks using these same neuroscience-inspired systems. For feasible integration with cutting-edge technology including autonomous cars, drones and brain machine interfaces, it is essential that these neural networks function under such low-power, small-size, and real-time speed constraints. In this work we describe a novel dynamically-reconfigurable array of Mihalas-Niebur neurons implemented in 500nm CMOS technology. This neural array was designed to minimize power and area per neuron while maintaining biological real-time speeds. We further will demonstrate how this system can be used in a visual processing task.

5.2 Related Work

Considering the typical high-power consumption and large size of CPUs and GPUs, we instead focus on the design of neuromorphic systems within VLSI technology. Current state-of-the-art large-scale neural arrays implemented in VLSI technology include the Neurogrid (Stanford University) [12], TrueNorth (IBM) [27], SpiN-

Naker (University of Manchester) [28], and BrainScales (University of Heidelberg) [26].

5.2.1 Neurogrid

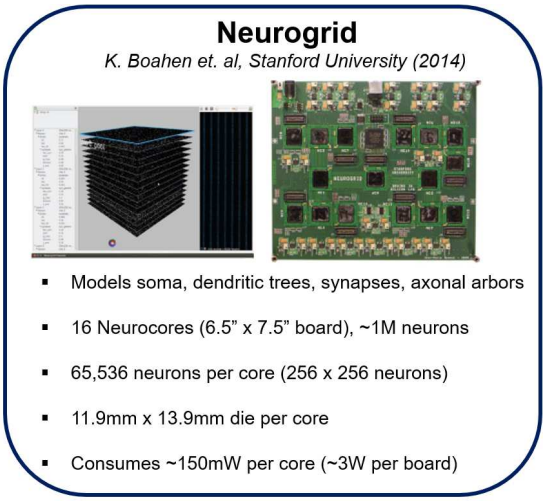


Figure 5.1: Neurogrid Overview [12].

Neurogrid is a mixed-mode multichip system primarily used for large-scale neural simulations and visualization [12]. It is depicted in Fig. 5.1. The neuron circuits used in Neurogrid are closely correlated to the physical characteristics of neurons in the brain. It models the soma, dendritic trees, synapses, and axonal arbors. It consists of 16 neurocores/chips each with 65k neurons (totaling 1 M neurons) implemented in subthreshold analog circuits. A single neurocore is fabricated on a 11.9mm $\times$ 13.9mm die. A board of 16 neurocores is of size 6.5" $\times$ 7.5" and the complete board consumes roughly 3W of power (a single neurocore consumes $\sim 150mW$).

5.2.2 BrainScales

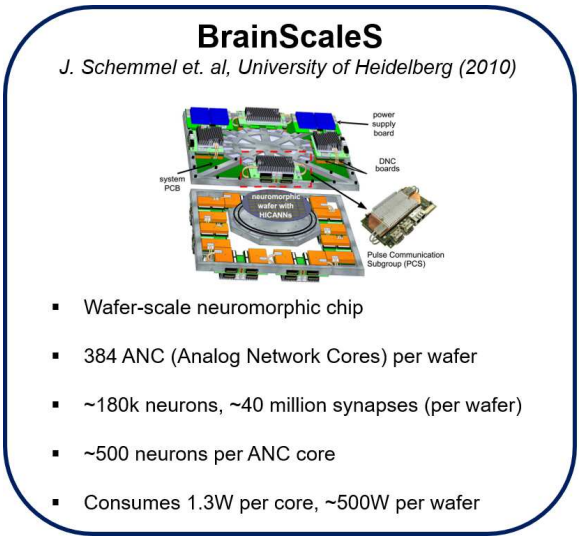


Figure 5.2: BrainScales Overview [26].

Also using mixed-mode VLSI circuits for neuron implementation is the BrainScales system [26]. BrainScales uses wafer-scale integration for implementing their neural array with efficient interconnectivity allowing for 10,000× faster speeds than biological systems. Each 20cm wafer consists of 44 reticles containing 8 HiCANN (High-Count Analog Neural Network) dies per reticle. Each HiCANN die contains 512 adaptive exponential integrate-and-fire neurons implemented in above-threshold circuits. This totals to 352 dies and $\sim 200k$ neurons and $\sim$ 40 M synapses on a single wafer. A single HiCANN die consumes 1.3W and a single wafer consumes $\sim 500W$. This BrainScales system as a whole is comprised of 20 of these wafers, compensating area and power for speed, allowing acceleration in simulation of long-term biological processes (i.e. learning).

5.2.3 TrueNorth

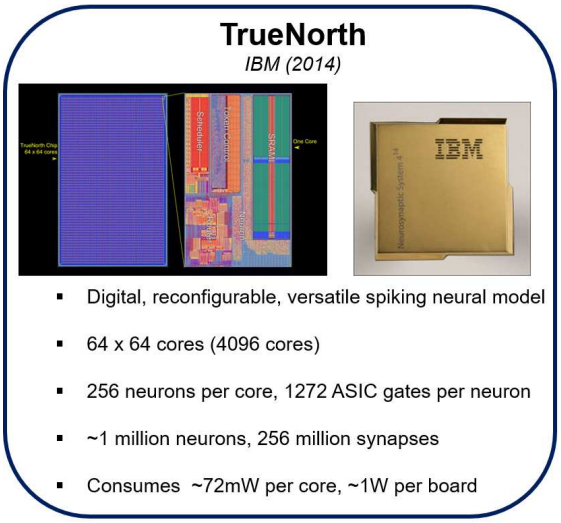


Figure 5.3: TrueNorth Overview [27].

IBMs TrueNorth neuromorphic chip consists of 1 million digital neurons capable of various spiking behaviors [27]. Each die holds 4096 cores, each core holding 256 digital neurons and 256 synapses per neuron. A single die consumes $\sim 72mW$ of power. They have developed a board (NS16e) comprised of 16 TrueNorth chips, consuming 1W of power at 1KHz speed making it ideal for energy-efficient applications. Although digital in its implementation, low-power consumption is due to fabrication in an aggressive, state-of-the-art 28nm technology process.

5.2.4 SpiNNaker

SpiNNaker [28], another digital neuromorphic neural array, was designed for scalability and energy-efficiency by incorporating brain-inspired communication methods.

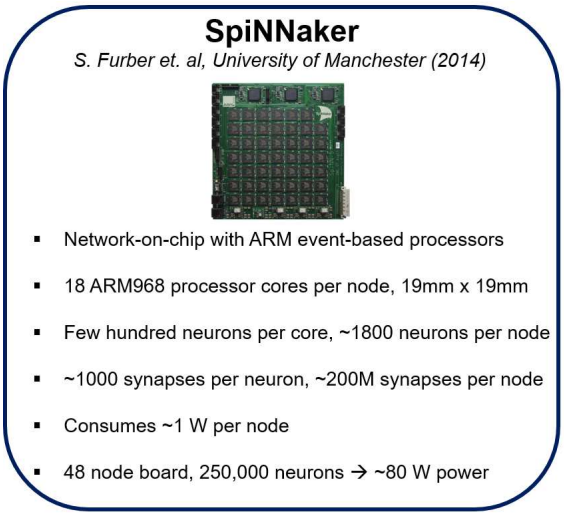


Figure 5.4: SpiNNaker Overview [28].

It can be used for simulating large neural networks and performing event-based processing for other applications. Each node is comprised of 18 ARM968 processor cores, each with 32 Kbytes of local instruction memory, 64Kbytes of local data memory, packet router, and supporting circuitry. A single node consists of 16,000 digital neurons consuming $\sim 1W$ of power per node. There exists two SpiNNaker circuit boards, the smaller being a 4-node (64,000 neurons) board and the largest being 48-node board (768,000 neurons). The 48-node board consumes $\sim 80W$ of power.

5.2.5 IFAT-inspired Systems

The work presented here is inspired by the integrate-and-fire array transceiver (IFAT). Although there exists various derivations, in its originality, the IFAT is an array of neurons with dynamic, reconfigurable synapses stored in a memory-based

look-up-table (LUT) external to the neuron array. It is implemented in mixed-signal VLSI technology and uses an Address Event Representation (AER) communication protocol. Using an AER receiver and transceiver, it allows events to be received and transmitted asynchronously in an event-based, time-multiplexed fashion. When the neuron array receives or transmits an address-event (AE), it corresponds to the receiving or transmitting of a spike (event), respectively. Originally, the IFAT was designed using integrate-and-fire neuron models with both probabilistic and conductance-based synapses [14, 77]. More recently, a 65k-neuron array using a two-compartment neuron cell with conductance-based synapses has been designed in Gert Cauwenberghs lab [78]. It consumes an area of 16 mm^2 in a 90nm CMOS technology. In this same lab, the Hierarchical Address-Event Routing (HiAER) architecture was designed for a neural array to achieve scalable communication of neural and synaptic spike events between neuromorphic processors [133]. Giacomo Indiveri's lab has also made significant contributions to the evolution of the IFAT. More recently, his group designed an array of 32 neurons with local, on-chip asynchronous SRAM for storing synaptic weights in 0.35μm technology [79]. Indiveris group also designed a reconfigurable on-line learning spiking (ROLLS) neuromorphic processor [134]. The neuron circuits consists of synapses with bi-stable, spike-based plasticity to achieve both short-term and long-term learning.

5.2.6 This Work

The neural arrays previously described and even more not discussed, demonstrate the evolution of neural arrays designed in VLSI technology. We present an array of Mihalas-Niebur neurons with dynamically-reconfigurable synapses implemented in 0.5 μm CMOS technology optimized for low-power, low-mismatch, and high-density. This neuron model has been shown to produce many various spiking behavior allowing this system to have a wide-range of applications. This neural array has two modes of operation: one is each cell in the array operates as independent leaky integrate-and-fire neurons, and the second is two cells work together to model the Mihalas-Niebur neuron dynamics. Depending on the mode of operation, this implementation consists of 2040 Mihalas-Niebur neurons or 4080 I&F neurons within a 3mm $\times$ 3mm area. Each I&F neuron cell consumes an area of 1495μm and dissipates 14pJ (estimated from simulation at 1.0V) to 360pJ (measured at 5.0V) of energy per synaptic event. We also introduce a unique design and functionality of the array itself that enables a further decrease in power-consumption, increase the number of neurons per mm^2 silicon area, and significantly reduced mismatch. Low mismatch is essential for proper operation of the neuron circuits and producing accurate results when used for spike-based processing. We finally utilize this neuron array for a visual processing task to demonstrate its applicability real-world systems.

5.3 Neuron Model

Selecting and implementing the appropriate neuron model is a vital component of this proposed work. Currently, the silicon neuron model used in the IFAT chip is based on the integrate-and-fire model [77,135–137]. This model consists of integrating charge onto a capacitance. When the voltage across this "membrane" capacitance reaches a set threshold, a spike/event is generated. Although optimized with respect to chip area and power consumption, these models based on integrate-and-fire neurons are limited in the number of spiking behaviors they can produce. An ideal system is capable of producing all 20 of the neuron spiking behaviors [138].

5.3.1 SOTA Neuron Models

The Hodgkin-Huxley model was originally implemented in silicon hardware by Mahowald and Douglas in 1991 [65] and later optimized in [139]. This was one of the first silicon neuron models. It is a low-level abstraction of a neuron in that it models the electrophysiology of the neuronal membrane. While this model can produce many the spiking behaviors, it consisted of a complex design that required setting many analog and digital biases to get the different spiking behaviors. This restricted the number of different spiking behaviors exhibited across multiple neurons simultaneously. Other neuron models include McCulloch and Pitts [140], Fitzhugh-Nagumo [141], Morris-Lecar [142], Hindmarsh-Rose [143], Wilson Polynomial [144],

Resonate-and-fire [145], Exponential integrate-and-fire [146], and Adaptive exponential integrate-and-fire [147]. Majority of these models use mathematical abstractions to model the dynamics of various neuron spiking behavior. However, these models either are not able to produce all of the 20 neuron spiking patterns that we seek to achieve or they do not contain variables and parameters with direct biological correlates.

Finally, the Izikevich proposed a model [148] which was eventually implemented in VLSI technology [149]. This model was capable of producing all neuron spiking behaviors. However, this model is purely equation-based and contains parameters which lack any direct biological correlates.

Our goal is to design systems which mimic biological systems in not only their functionality but also their implementation. We seek to design a VLSI-based neuron model capable of producing all 20 neuron spiking behaviors and consists of parameters/variables with biological correlates. The Mihalas-Niebur model [150], proposed in 2009, is capable of producing all of these spiking behaviors and contains the biophysical relations that we seek. We propose to use this model in implementing our complete IFAT system.

5.3.2 Mihalas-Niebur Neuron

Each neuron in this neural array models the Mihalas-Niebur neuron dynamics [150]. In its original form, it uses linear differential equations and parameters with

biological facsimiles. It consists of an adaptive threshold and was shown to be capable of modeling all of the biologically-relevant neuron behaviors. It uses three differential equations modeling the internal currents (I'_j), membrane voltage (V'_m), and adaptive threshold voltage (Θ') dynamics:

$$I'_j(t) = -k_j I_j(t); j = 1, ..., N \tag{5.1}$$

$$V'_m(t) = \frac{1}{C}(I_{ext} + \sum_j I_j(t) - G(V_m(t) - E_L)) \tag{5.2}$$

$$\Theta'(t) = a(V_m(t) - E_L) - b(\Theta(t) - \Theta_\infty) \tag{5.3}$$

where I_j spike-induced currents, C is the membrane capacitance, I_{ext} is the external input current, k_j, G, E_L, , Θ_∞, a, and b are free variables. When the membrane voltage ($V_m(t)$) exceeds the threshold voltage ($\theta(t)$, the membrane voltage and threshold voltage are updated according to the following rules:

$$I_j(t) \rightarrow R_j \times I_j(t) + A_j \tag{5.4}$$

$$V_m(t) \rightarrow V_r \tag{5.5}$$

$$\Theta(t) \rightarrow max(\Theta_r, \Theta(t)) \tag{5.6}$$

where R_j and A_j are free parameters and V_r and Θ_r is the membrane and threshold reset voltage, respectively.

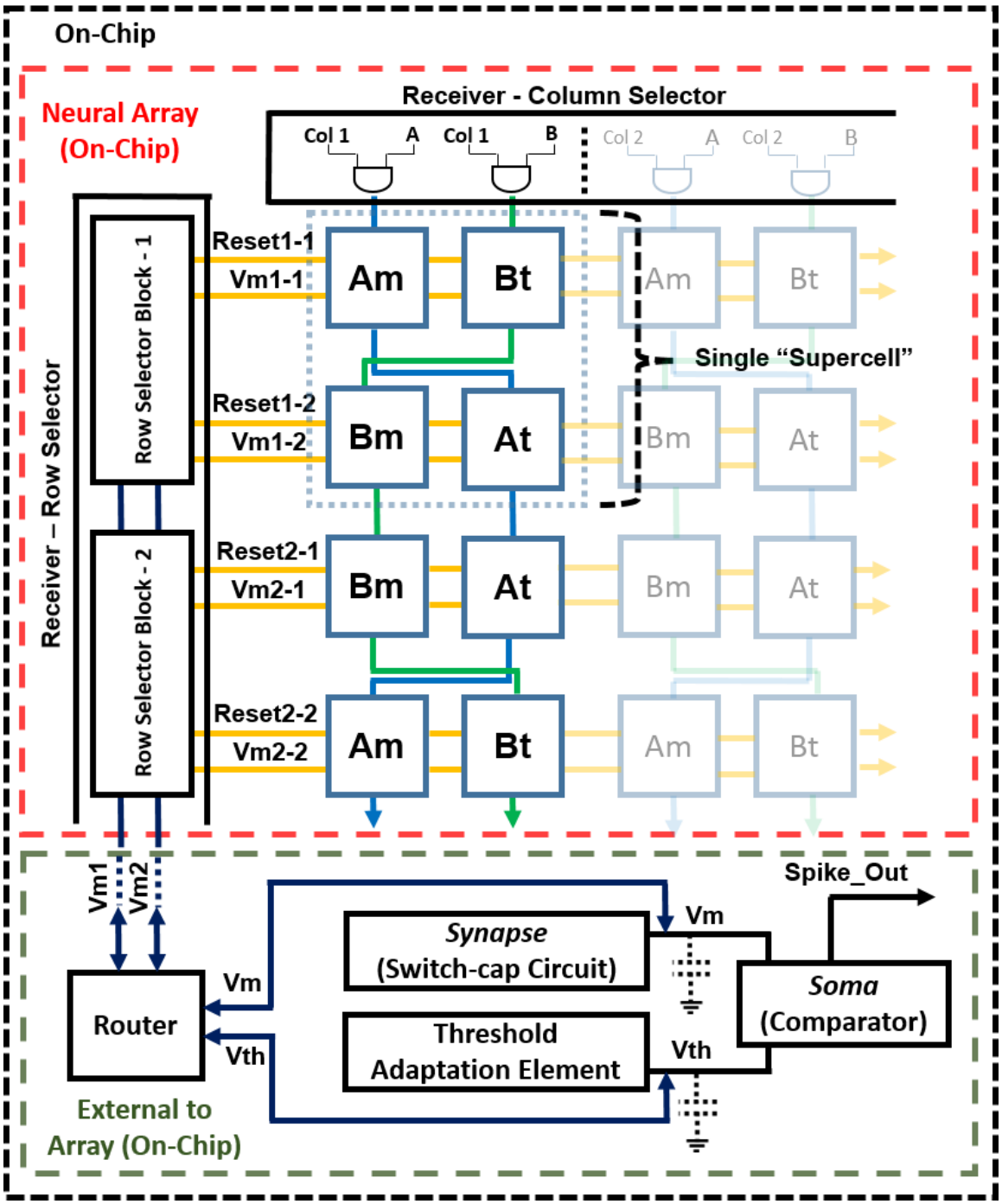


Figure 5.5: Mihalas-Niebur Neural Array Transceiver Block Diagram. Image from [29].

5.4 Chip Architecture

5.4.1 Neural Array

The full neuron array chip block diagram can be seen in Fig. 5.5. It was implemented with means to maximize the neuron array density, minimize power consumption, and reduce mismatch due to process variation. This is achieved by utilizing a single membrane synapse (switch-capacitor circuit) and soma (comparator) shared by all neurons in the array. The connections between neurons is reconfigurable via an off-chip look-up table (LUT). Presynaptic events are sent first through the LUT where the destination addresses and synaptic strengths are stored. Post-synaptic events are then sent to the chip. These events are sent as address-events (AE) along a shared address bus decoded by the row decoder and column decoder on-chip. The incoming address corresponds to a single neuron in the array.

The neuron array is made up of supercells, each containing four cells labeled, Am, At, Bm, and Bt. Each supercell contains two Mihalas-Niebur (M-N) neurons, one using Am and At cells, and the second using Bm and Bt cells. Each of these M-N neurons can also operate as two independent leaky integrate-and-fire (I&F) neurons resulting in a total of four leaky I&F neurons (Am, At, Bm, and Bt). Incoming AE select the supercell in the array and also consists of two additional bits for selecting one of the two M-N neuron (A or B) within the supercell, or one of the four cells when operating as I&F neurons. Finally, the voltage across the storage capacitance

for both the membrane cell and threshold cell is buffered to the processor via the router ($Vm1$-X and $Vm2$-X, where X is the row selected). The router is used for selecting which voltage (from the membrane cell or threshold cell) is buffered to the processor as the membrane voltage and/or threshold voltage, depending on the mode selected (M-N mode or I&F mode). This router is necessary for allowing the voltage from the threshold (At or Bt) cell to be used as the membrane voltage when in I&F mode. After the selected neuron cell(s) buffer its stored voltage to the external capacitances C_m and C_t, the synaptic event is applied and the new voltage is buffered back to the same selected cells that received the event. The synapse and threshold adaptation elements execute the neuron dynamics as events are received. If the membrane voltage exceeds the threshold voltage, there is a single comparator (soma) that outputs a logic high (event).

An output arbiter/transmitter is not necessary in our design considering that a neuron only fires when it receives an event. The single output signal always corresponds to the neuron that receives the incoming event. Having a single comparator not only reduces power consumption but also reduces the required number of pads for digital output. In this design we compromise speed (for low-power and low-area) due to the time necessary to read and write to and from the neuron. However, we are still capable of achieving a maximum input event rate of $\sim$1 MHz for proper operation.

5.4.2 Circuit Implementation of Mihalas-Niebur Neuron

Each cell pair (Am/At and Bm/Bt) in this neural array models the Mihalas-Niebur neuron dynamics [150]. The M-N neuron uses linear differential equations and parameters with biological facsimiles. It consists of an adaptive threshold and was shown to be capable of modeling all of the biologically-relevant neuron behaviors. It uses differential equations modeling the internal currents, membrane voltage, and adaptive threshold voltage dynamics. Update rules are applied for each time the membrane voltage exceeds the adaptive threshold voltage [150]. For circuit implementation of this M-N model, we make a few modifications. The first is omitting internal induced-spike currents, and the second is setting the reset voltage equal to the resting potential. We make these modifications at the expense of the generality of the model. However, this modified M-N model is still capable of implementing 9 of the biologically-relevant spiking behaviors [30]. The modified differential equations for CMOS implementation are as follows:

$$V'm(t) = \frac{g_l^m}{C_m}(V_r - V_m(t)) \tag{5.7}$$

$$\theta'(t) = \frac{g_l^t}{C_t}(\theta_r - \theta(t)) \tag{5.8}$$

$$V_m(t+1) = V_m(t) + \frac{C_s^m}{C_m}(E_m - V_m(t)) \tag{5.9}$$

$$\theta(t+1) = \theta(t) + \frac{C_s^t}{C_t}(V_m(t) - V_r) \tag{5.10}$$

and,

$$g_l^{m,t} = \frac{1}{r_l^{m,t}} = f_l^{m,t} C_l \tag{5.11}$$

This modification allows the use of two neuron cells to work together to model a single M-N neuron. This will be further discussed in proceeding sections. Eq. (5.9) and (5.10) model the change in membrane potential (V_m) and threshold potential (θ) at each time step as the neuron receives an input. C_s^m and C_s^t are the switch-capacitor capacitance depicting the synapse conductance or threshold adaptation conductance, respectively. Cm and Ct are the storage capacitance for the membrane and threshold cells, respectively. E_m is the synaptic driving potential. Eq. (5.7) and (5.8) model the leakage dynamics, independent of synaptic connections. $g_l^{m,t}$ are the leakage conductances for the membrane and threshold and are dependent on the clock frequency, $f_l^{m,t}$. The update rules for this modified M-N neuron model are as follows:

$$V_m(t) \leftarrow V_r \tag{5.12}$$

$$\theta(t) \leftarrow \begin{cases} if\ \theta(t) > V_m(t), \theta(t) \\ if\ \theta(t) \leq V_m(t), \theta_r(t) \end{cases} \tag{5.13}$$

As previously noted, the synapse dynamics (switch-cap circuits), soma (comparator), and supporting circuits are on-chip, but external to the neuron cells. The neuron cell consists of only the storage capacitance, leakage dynamics, and buffer, coupled with transmission-gate based switches for reading and writing to and from the neuron (See 5.6). One cell is used for modeling the membrane dynamics while a second cell is used

for modeling the adaptive threshold dynamics. There is also the option to model each neuron as an independent leaky I&F neuron.

5.4.2.1 Neuron Cell

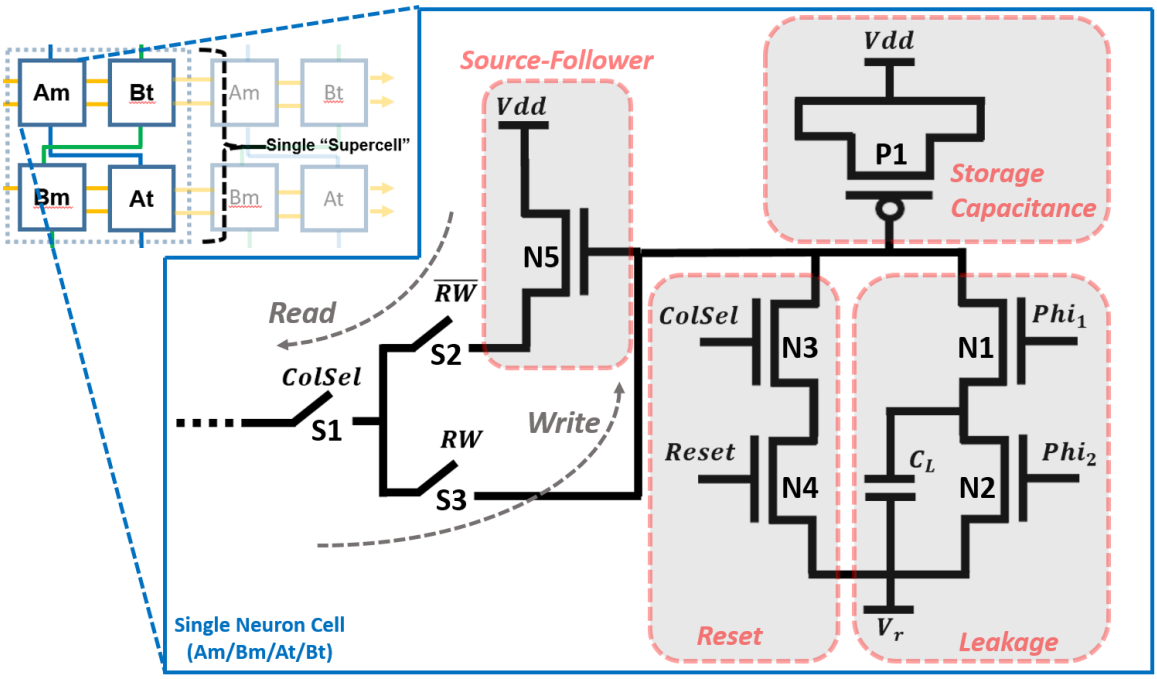


Figure 5.6: Neuron cell schematic.

The PMOS transistor, $P1$, is the storage capacitance (∼440fF), C_m or C_t, (depending on whether the cell is being used to model the membrane or threshold dynamics) implemented as a MOS capacitor with its source and drain tied to Vdd. Transistors $N1$ and $N2$ model the leakage (Eq. (5.7) and (5.8)) via a switch-capacitor circuit with Phi_1 and Phi_2 pulses at a rate of $f_l^{m,t}$ (also $C_L << C_m$). Transistors $N3$ and $N4$ allow for resetting the neuron when selected ($ColSel = 1$). Transistor $N5$ forms

a source-follower when coupled with a globally-shared variable resistance located in the processor of the neural array. It is implemented as an NMOS transistor with a voltage bias (Vb). In read mode ($RW = 0$), switch $S2$ is closed such that the voltage across the storage capacitance is buffered to an equivalent capacitance coupled to the synapse and/or threshold adaptation element. In write mode ($RW = 1$), switch $S3$ is closed such that the new voltage from the synapse/threshold elements (after an event is received) is buffered to the storage capacitance. The layout of the neuron cell can be seen in Fig. 5.7.

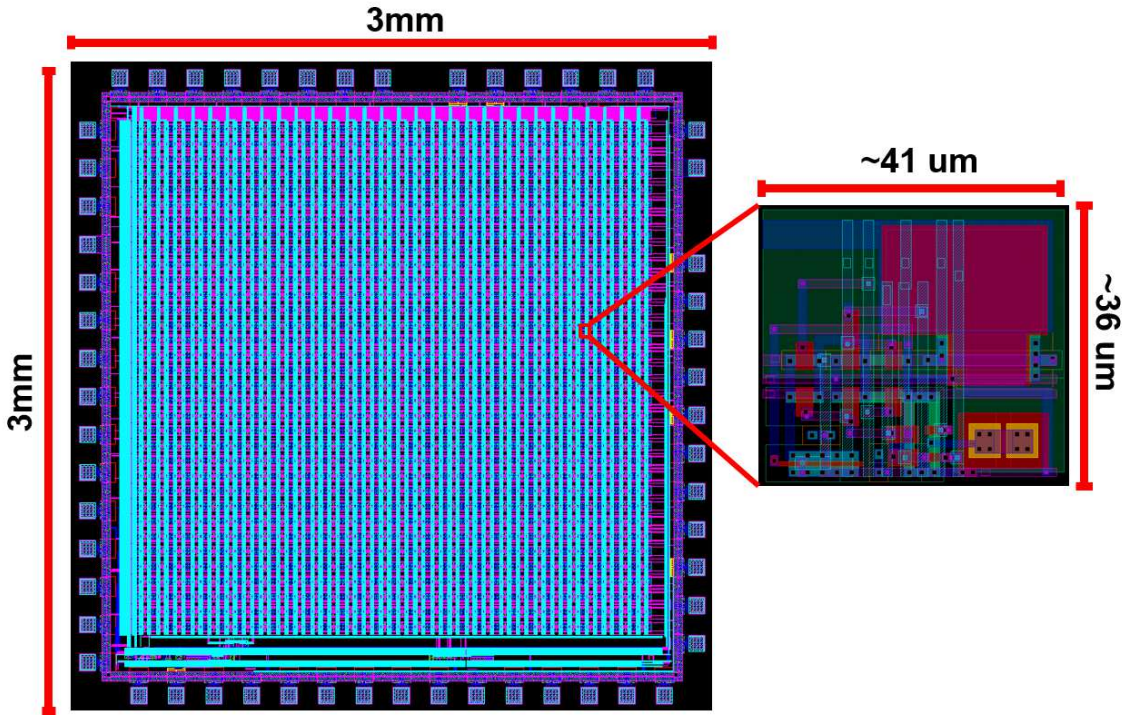


Figure 5.7: Neuron cell layout.

5.4.2.2 Synapse and Threshold Adaptation

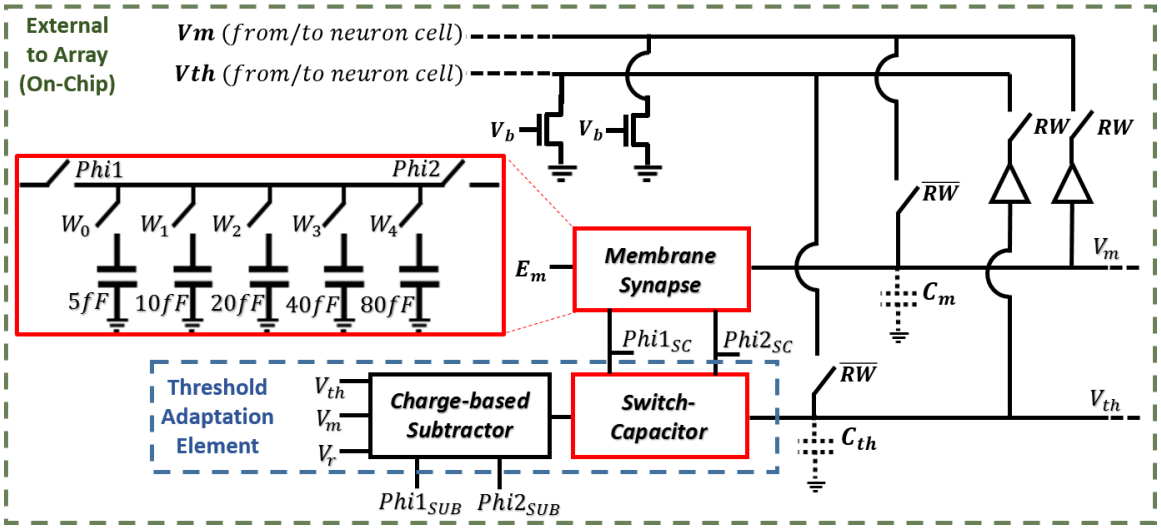


Figure 5.8: Single, shared, on-chip synapse and threshold adaptation schematic external to the neural array.

The schematic for modeling the neuron dynamics can be seen in Fig.5.8. When a neuron receives an event, $RW = 0$, and the neuron's cell is selected and its stored membrane voltage is buffered to the capacitance C_m. In the same manner, if in M-N mode, the threshold voltage is buffered to C_t. The $Phi1_{SC}$ and $Phi2_{SC}$ pulses are then applied (off-chip), adding (excitatory event) or removing (inhibitory event) charge to C_m via the synapse using a switch-capacitor circuit. A second, identical switch-capacitor circuit is used for implementing the threshold adaptation dynamics. As a neuron receives events, the same $Phi1_{SC}$ and $Phi2_{SC}$ pulses are applied to the threshold adaptation switch-capacitor circuit which adds or removes charge to C_t. The new voltage is then buffered ($RW = 1$) back to the neuron cells for storing the

new membrane voltage (as well as the threshold voltage if in M-N mode). When using each neuron independently as leaky I&F neurons, the threshold adaptive element is bypassed and an externally applied fixed threshold voltage is used. A charge-based subtractor is used in the threshold adaptation circuit for computing $V_{th} + (V_m - V_r)$ in modeling Eq.5.10. This subtraction output is the driving potential for the threshold switch-capacitor circuit. An externally applied voltage, E_m, is the synaptic driving potential for the membrane synapse and is used for modeling Eq.5.9. Finally, the comparator outputs an event when the membrane voltage exceeds the threshold voltage. An external reset signal for both the neuron cell modeling the membrane voltage and cell modeling the threshold voltage is activated for the selected neuron (via $Reset1\text{-}X$ and $Reset2\text{-}X$) when a spike is outputted. The layout location can be seen in Fig. 5.9.

5.4.3 AER - Row and Column Select

The AER row and column receiver circuits are simple row and column decoders. The row and column decoders are responsible for selecting a single I&F neuron or Mihalas-Niebur neuron. The column decoder selects a single supercell column while the row decoder selects either a single I&F neuron from that supercell by selecting either A or B neuron within the supercell. There is no handshaking implemented for this receiver. Fig. 5.10 shows the layout location of this row and column decoder used as the AER receiver. It is important to note that there is no transmitter. This

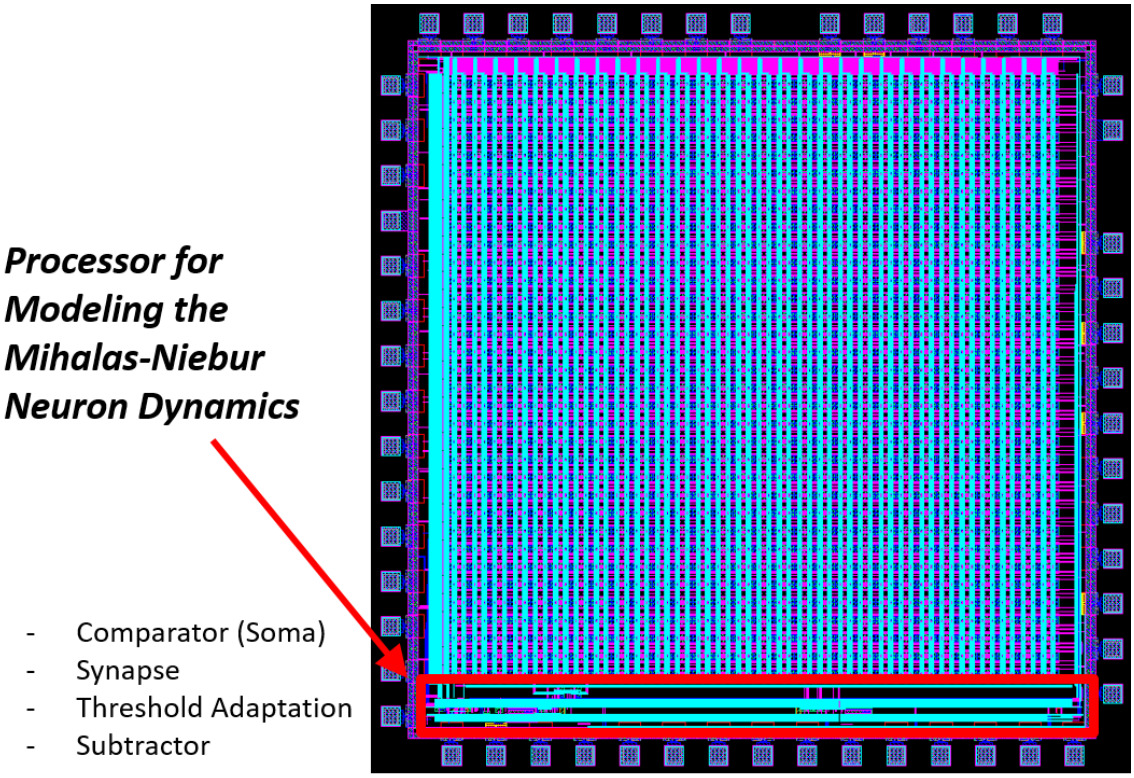


Figure 5.9: Layout location for modeling the Mihalas-Niebur neuron dynamics.

is because there is only one comparator, and therefore, one output which is only triggered when a neuron receives an event.

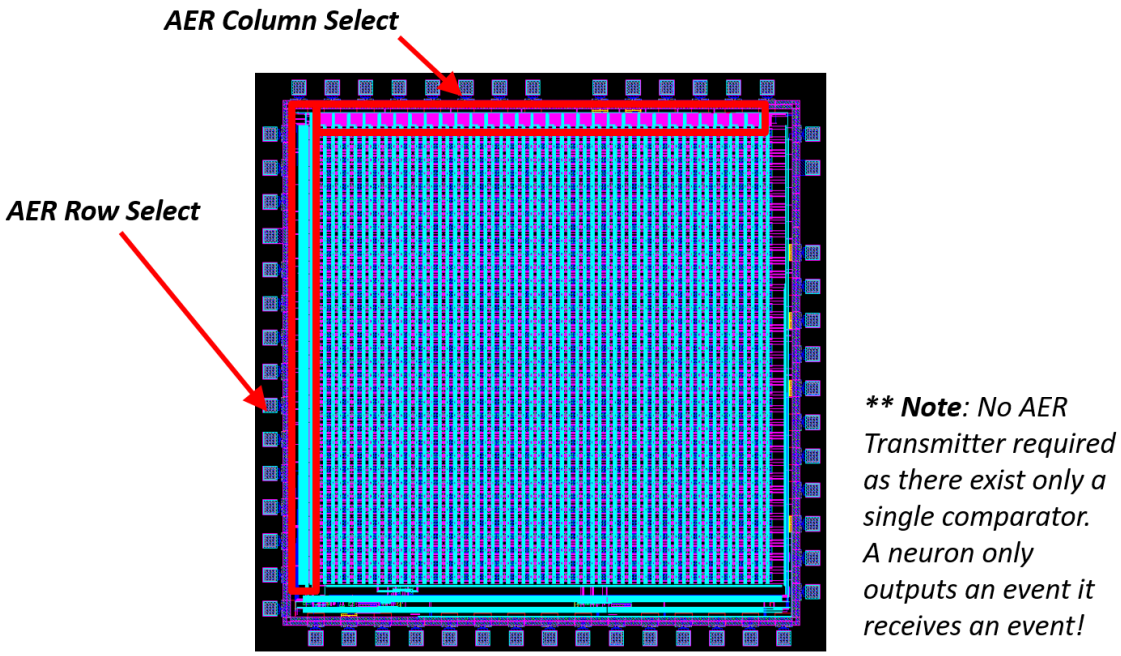


Figure 5.10: Layout location for row and column decoder used as the AER receiver.

5.4.4 Router

There also exists a router circuit which is made up of switches implemented as transmission gates for selecting the appropriate membrane and threshold voltages to be passed through to the external processor circuits (i.e. synapse, threshold adaption). It is controlled by signals such as “simpleMode” and “useTestNeuron” for controlling whether to buffer the threshold/membrane voltages from the testNeuron or external biases, etc.

5.4.5 Subtractor

A charge-based subtractor is used within the circuitry external to the array for implementing the Mihalas-Niebur neuron dynamics. It is used for computing:

$$V_{th} + (V_m - V_r) \tag{5.14}$$

This is output is used as the driving potential of the threshold adaptation switch-cap circuit. This allows for the threshold adaptation dynamics. As the membrane voltage increases, the output of the subtractor increases which in turns increases the amount of charge accumulated onto the threshold storage capacitance. This increases the threshold voltage as the membrane voltage increases allowing for spike frequency adaptation.

5.4.6 Representing Excitatory and Inhibitory Events

Excitatory events are represented as positive spikes such that events from an excitatory synapse increase the membrane voltage (See Fig. 5.17 and Fig. 5.18). An excitatory event occurs when the synaptic driving potential, E_m is greater than the membrane voltage, V_m. As events are received, the membrane voltage will increase towards the E_m voltage. The rate at which it increases toward E_m is further controlled by the digital weight W_m. A higher value of W_m, the faster the membrane voltage will increase toward E_m as events are received.

To represent inhibitory events, E_m must be set to a voltage less than V_m. These are negative events such that charge is removed from the membrane capacitor and the membrane voltage, V_m, decreases as events are received via this inhibitory synapse. Similarly to excitatory events, the membrane voltage will tend toward E_m as events are received in a decreasing fashion. This can be visualized in Fig. 5.19.

5.4.7 Chip I/O

The pinout of the chip can be seen in Table 5.1.

Type	**Name**	**Description**	**Pin #**
N/C	N/A	N/A	1
Digital Input	CS	Chip select allowing for column selection.	2
Digital Input	useTestPix	Bypass array and use test neuron only.	3
Digital Input	tClk	Threshold cell leakage clock.	4
Digital Input	mClk	Membrane cell leakage clock.	5
Analog Input	Vrt	Threshold cell reset voltage.	6
Analog Input	Vrm	Membrane cell reset voltage.	7
Digital Input	ColAddr6	AER receiver column address bit 5.	8

Digital Input	ColAddr5	AER receiver column address bit 4.	9
Digital Input	ColAddr4	AER receiver column address bit 3.	10
Digital Input	ColAddr3	AER receiver column address bit 2.	11
Digital Input	ColAddr2	AER receiver column address bit 1.	12
Digital Input	ColAddr1	AER receiver column address bit 0.	13
Digital Input	AddrX	Select column/neuron A/B ("0" = A).	14
Digital Input	SelAllCol	Select all columns.	15
N/C	N/A	N/A	16
N/C	N/A	N/A	17
Digital Input	Rm	Membrane cell reset signal.	18
Digital Input	Rt	Threshold cell reset signal.	19
Digital Input	SelAllRow	Threshold cell reset signal.	20
Ground	GND	Ground.	21
Digital Input	RowAddr6	AER receiver row address bit 5.	22
Digital Input	RowAddr4	AER receiver row address bit 3.	23
Digital Input	RowAddr5	AER receiver row address bit 4.	24
Digital Input	RowAddr3	AER receiver row address bit 2.	25
Digital Input	RowAddr1	AER receiver row address bit 0.	26
Digital Input	RowAddr2	AER receiver row address bit 1.	27

Digital Input	Phi1p	First pulse (ϕ_1) for charge-based subtractor.	28
Digital Input	Phi2p	Second pulse (ϕ_2) for charge-based subtractor.	29
Digital Input	Phi3p	First pulse (ϕ_1) for membrane and threshold switch-cap.	30
Digital Input	Phi4p	Second pulse (ϕ_2) for membrane and threshold switch-cap.	31
Digital Input	thrSwitch	Switch between Vm (="0") and Vt (when in simple mode).	32
N/C	N/A	N/A	33
Digital Input	simpleMode	Select between simple/I&F (="1") and adaptive/M-N mode.	34
Analog Input	VbS	Bias for subtractor op-amp.	35
Digital Input	Precharge	Precharge signal for Vm and Vt wires to and from processor.	36
Analog Output	Et	Current synaptic driving potential for threshold switch-cap.	37
Digital Input	W0t	Weight of threshold switch-cap bit 0.	38

Digital Input	W1t	Weight of threshold switch-cap bit 1.	39
Digital Input	W2t	Weight of threshold switch-cap bit 2.	40
Digital Input	W3t	Weight of threshold switch-cap bit 3.	41
Digital Input	W4t	Weight of threshold switch-cap bit 4.	42
Analog Input	Em	Synaptic driving potential for membrane synapse.	43
Digital Input	W0m	Weight of membrane switch-cap/synapse bit 0.	44
Digital Input	W1m	Weight of membrane switch-cap/synapse bit 1.	45
Digital Input	W2m	Weight of membrane switch-cap/synapse bit 2.	46
Digital Input	W3m	Weight of membrane switch-cap/synapse bit 3.	47
Digital Input	W4m	Weight of membrane switch-cap/synapse bit 4.	48

N/C	N/A	N/A	49
Analog Output	Vm	Current membrane voltage of selected neuron.	50
Analog Output	Vt	Current threshold voltage for selected neuron.	51
Analog Input	VbP	Bias voltage for buffers in analog output pads.	52
Digital Output	Out	Spike output signal.	53
Digital Input	thrCheck	Check if $Vt > Vrm$ before updating Vt (= "1").	54
Digital Input	SetEt	Selects $Etin$ (when = "1") or subtractor output for Et.	55
Analog Input	VbC1	Bias voltage for threshold check comparator.	56
Analog Input	EtIn	Threshold switch-cap synaptic driving potential.	57
Analog Input	VbC2	Bias voltage for spike output check comparator.	58
Digital Input	RW	Read/Write signal for reading to and writing from neuron.	59

Analog Input	VbSF	Bias voltage for read/write buffer (source-follower).	60
Power	VDD	Power (nominal VDD = 5.0 V).	61
Analog Input	VDD	Threshold voltage input when in simple mode	62
N/C	N/A	N/A	63
N/C	N/A	N/A	64

Table 5.1: Mihalas-Niebur Neuron Array Input/Output

5.5 PCB Architecture

An image of the printed circuit board (PCB) can be seen in Fig. 5.11. This is a 4 layer board (top, bottom, power, and ground planes) and is 3.5" × 2.5" in size. A high-level block diagram of the complete system implemented on the board can be seen in Fig. 5.11.

Events are generated by either a PC, event-based camera such as ATIS [16] or DVS [15], or any other event-based system. These events are considered pre-synaptic events and are transmitted to the FPGA. The board used is an Opal Kelly 6010-LX150. It consists of a Spartan-6 FPGA. Using block RAM on the FPGA, a look-up-table (LUT) is implemented. A state-machine on the FPGA controls the communication

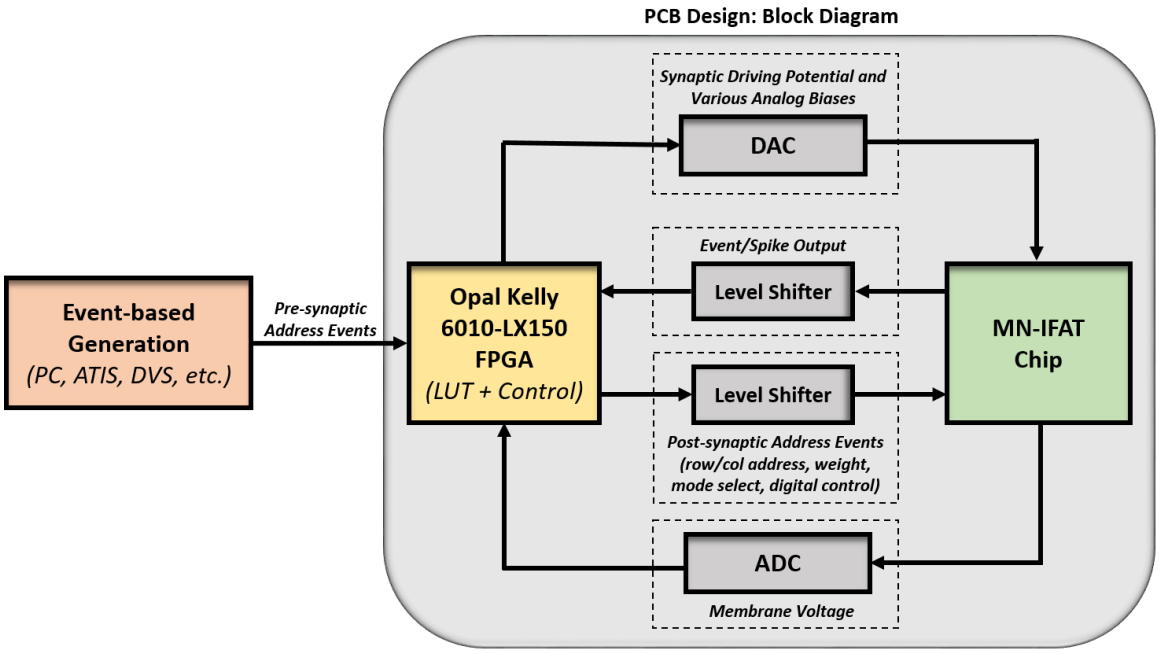


Figure 5.11: Block diagram of complete system Mihalas-Niebur Neural Array System.

of the incoming address event with the LUT. The post-synaptic address events for each incoming address event are obtained along with their corresponding synaptic weight. This synaptic weight is a function of both the digital weight $Wm < 0 : 4 >$ and Em potential. The values $Wt < 0 : 4 >$ and $Etin$ are also stored in the LUT an obtained when using adaptive mode. The digital representation of Em and Et are sent to the DAC which output voltage biases Em and Et to the chip. There exist DACs for outputting voltage biases for the comparator and op-amps on-chip. When address events are transmitted to the chip, eventually neurons may fire and output a spike/event which is registered by the FPGA. Level shifters are necessary for communicating between the FPGA (operating at 3.3V) and chip (operating at 5.0V). Finally, the membrane voltage output from the chip can be realized via an

ADC for observing the Vm potential digitally.

5.5.1 FPGA Integration and Control

The FPGA is responsible for controlling interaction between the event generator and the chip including memory allocation for the LUT. The complete block diagram of the system implemented on FPGA can be seen in Fig. 5.12. The FPGA consists of 4 main components:

1. Input FIFO - Store incoming/presynaptic address events
2. LUT BRAM - Block RAM for storing LUT
3. Output FIFO - Store outputted address events (from chip)
4. State Machine - Control interactions to and from chip

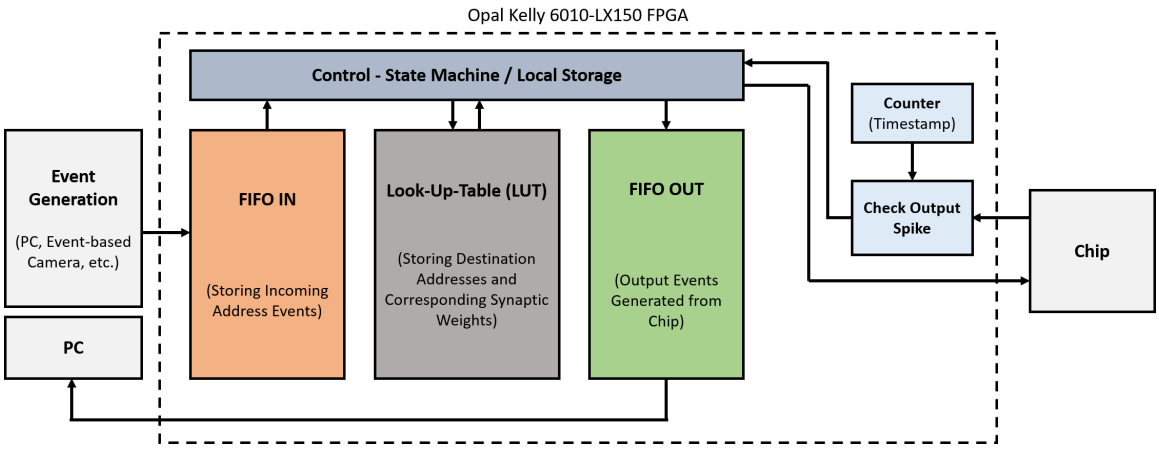


Figure 5.12: Block diagram of FPGA system used for controlling chip.

5.5.1.1 Input FIFO and AE Representation

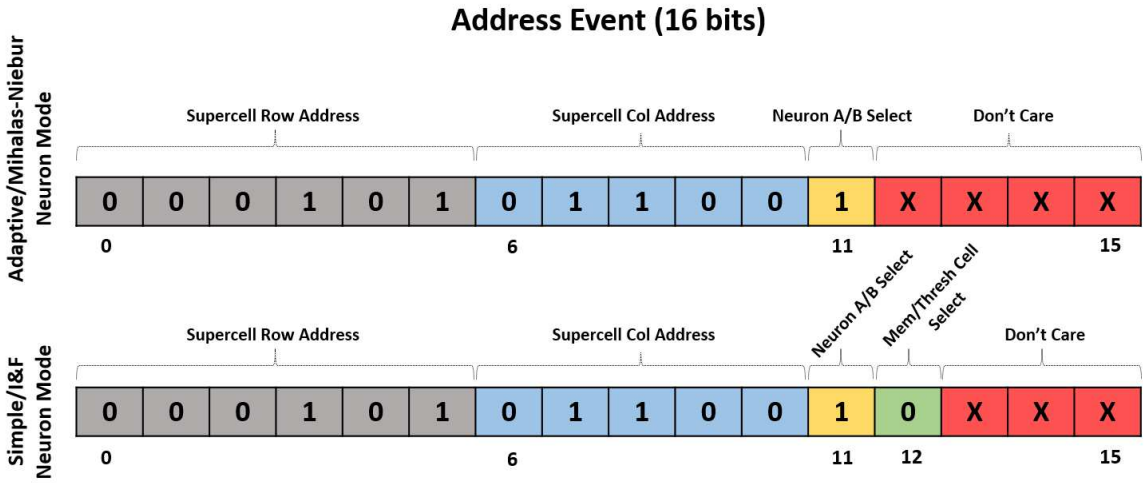


Figure 5.13: Presynaptic address event representation for both adaptive and I&F modes.

The input FIFO (First-In First-Out) is responsible storing incoming presynaptic address-events (See Fig. 5.13). The width of each address event is 16 bits. The row and column addresses select a single supercell. There are 34 rows and 30 columns of supercells. As previously noted, each supercell consists of 2 Mihalas-Niebur neurons (A and B) or 4 I&F neurons (Am, At, Bm, and Bt). Address-events are represented differently depending on the mode of operation (I&F or Mihalas-Niebur/adaptive). The first 11 bits of the incoming address event is allocated to row and column address of the supercell (6 bits for row, 5 bits for column). Bit 12 is allocated for selecting neuron A (= "0") or B (="1"). This is all of the address information required for adaptive mode. However, in simple mode (I&F), selection between the membrane

cell (Am/Bm) or threshold cell (At/Bt) must be specified. Therefore, in this case, the bit 13 is allocated for selecting the membrane or threshold cell within neuron A (or B). The remaining bits are "Don't Care" bits.

5.5.1.2 Look-Up-Table

The look up table consists of the destination addresses for each presynaptic address event. A diagram of the look-up-table can be seen in Fig. 5.14. For each destination address, there is a corresponding synaptic weight represented by $Wm < 0 : 4 >$, Em, $Wt < 0 : 4 >$, and $Etin$. However, in simple mode $Wt < 0 : 4 >$ and $Etin$ are not used. Memory is still allocated for these signals. Each destination address event requires a maximum of 13 bits (13 bits in simple mode, or 12 bits in adaptive mode) for storing the address of the selected neuron to which the event should be sent. The weight requires 5 bits for Wm, 5 bits for Wt, 8 bits for Em and 8 bits for $Etin$ (8-bit precision used for voltage biases). This results in 13 bits for address and 26 bits for its corresponding weight, totaling to 39 bits for each synapse. In simple mode, $Etin$ and Wt are "Don't Care" bits.

5.5.1.3 Output FIFO

When a neuron outputs a spike, its address is registered on the FPGA and loaded into the output FIFO along with its timestamp. The output FIFO has a width equal to that of the input FIFO (16 bits). The address event is represented in the

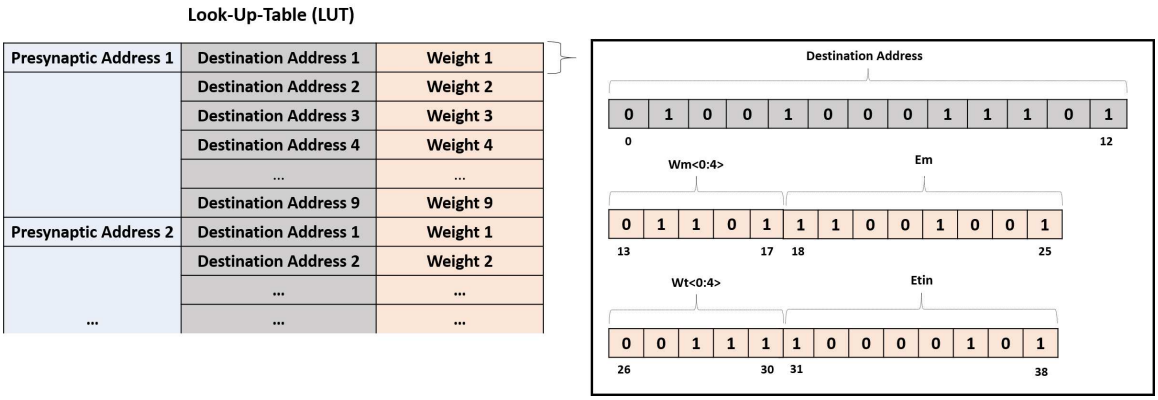


Figure 5.14: Look-up-table memory allocation for configuring synaptic connections with corresponding weights.

same manner as that of the presynaptic address event stored in the input FIFO (See Fig. 5.13). The output address event along with a 16-bit timestamp is loaded into the FIFO sequentially every time an event is outputted by a neuron. A 16-bit counter (0 to $2^{16} - 1 = 65535$) continuously counts upward. The timestamp is the current value of this counter at the time an output event occurs. This counter overflows every 65535 clock cycles. Therefore, it is necessary to output a flag when an overflow has occurred. This flag is represented by an address event loaded into the output FIFO which is outside the range of the neuron array. This signals that there was an overflow. The clock controlling this counter is the output from a clock divider to allow for control of the counter speed.

5.5.1.4 State Machine for Control

A state machine is used for controlling the interaction with the LUT and sending an event to a selected neuron on the chip. The state machine sequential steps necessary for sending an event when a presynaptic event is received can be seen below:

1. Wait for input FIFO to contain an address event.
2. Load next presynaptic address event (1 CC).
3. Load destination address event and corresponding weight (1 CC).
4. Register digital weight and wait for ADC to output voltage (25 CC).
5. Hold RW signal low for reading current membrane voltage of neuron to processor (10 CC).
6. Assert $Phi1p$ high (3 CC).
7. Assert $Phi1p$ low (1 CC).
8. Assert $Phi2p$ high (3 CC).
9. Assert $Phi2p$ low (1 CC).
10. Assert $Phi3p$ high (3 CC).
11. Assert $Phi3p$ low (1 CC).
12. Assert $Phi4p$ high (3 CC).

13. Assert *Phi4p* low (1 CC).

14. Set *RW* signal high for writing result back to membrane capacitance of neuron (10 CC).

15. Check for output spike (1 CC).

16. If output spike is high, write address event and timestamp to output FIFO (4 CC).

17. If output spike is high, reset neuron (25 CC).

This totals to a maximum of 93 clock cycles for sending a single event. In this system, we utilize a 100 MHz clock. Therefore, the max event rate is $\sim$ 1 1.07 MHz (See Equation 5.15).

$$\texttt{Max Event Rate} = \frac{1}{93 \times \frac{1}{100e6}} \approx 1.07 \texttt{ Mevents/sec} \tag{5.15}$$

5.6 Results and Discussion

In these results section we discuss simulation results validating the ability to produce 9 of the 20 biologically-relevant spiking behaviors. We then discuss experimental results and image processing applications using this novel neural array.

5.6.1 SPICE Simulation

We validate the mathematical model of this circuit implementation see in Section 5.4.2 using a Python implementation of this model. The results of the spiking behaviors it is capable of producing can be seen in Fig. 5.15. The parameters used for simulation are listed in Table 5.2. The model can implement 9 of the 20 biologically relevant neuron behaviors. For exhibiting tonic spiking and class 1 behaviors, the threshold voltage is maintained at a constant value. For all the other behaviors, threshold adaptation of the neuron plays a major role in generating action potentials (spikes).

Fig. 5.16 shows the SPICE simulation results based on the designed circuits. Table 5.3 shows a comparison of the design presented in this paper along with other relevant work based on the same Mihalas-Niebur neuron model. The design presented here occupies the lowest area and consumes the lowest power.

5.6.2 Experimental

5.6.3 Neuron Area

A single neuron cell in this array has dimensions of $41.7\mu m \times 35.84\mu m$. A comparison to other state-of-the-art neural array chips can be seen in Table 5.5. It consumes only 62.3% of the area consumed by a single neuron cell in [77], also designed in a $0.5\mu m$ process. While we achieve 668.9 I&F neurons/mm^2, [77] achieves only 416.7

Table 5.2: Neuron Parameters for Simulation

Feature	f_l^m **(Hz)**	f_t^l **(Hz)**	C_m^s **(fF)**	C_t^s **(fF)**	V_r **(V)**	θ_r **(V)**	E_m **(V)**
Tonic spiking	10 k	0.2 M	175	0	1	4	[0, 5]
Phasic spiking	3 M	200	175	30	1	3	[0, 5]
Freq. Adaptation	10 k	0.2 M	175	30	1	3	[0, 5]
Class 1	1	300	175	0	1	3.5	[0, 3.51]
Rebound spike	10 k	30 k	175	60	2	4	[2, 0, 2]
Threshold variability	10 k	0.3 M	175	75	2	4	[2, 5, 2, 0, 2, 5, 2]
Accommodation	1 M	3 M	175	75	1	2.7	[1, 5, 1, 2.3, 3.6, 5, 1]
Integrator	0.1 M	3 M	175	20	1	3.3	[1, 5, 1, 5, 1, 5, 1, 5, 1]
Input bistability	10 k	30 k	175	20	1	3	[1, 4, 1, 4, 1]

Table 5.3: Comparison of Mihalas-Niebur Neuron Circuit Implementations

Design	**Results**	**Process**	**Area**	**Power**	**Behaviors**
This work	SPICE	0.5 μm	0.001 mm^2	0.38 nW	9
[151]	On chip	0.5 μm	0.05 mm^2	40 nW	10
[152]	SPICE	0.15 μm	0.005 mm^2	7.5 nW	15
[153]	SPICE	0.5 μm	0.04 mm^2	40 μW	Only 6 shown
[154]	On chip	0.5 μm	0.04 mm^2	70 μW	Only 6 shown

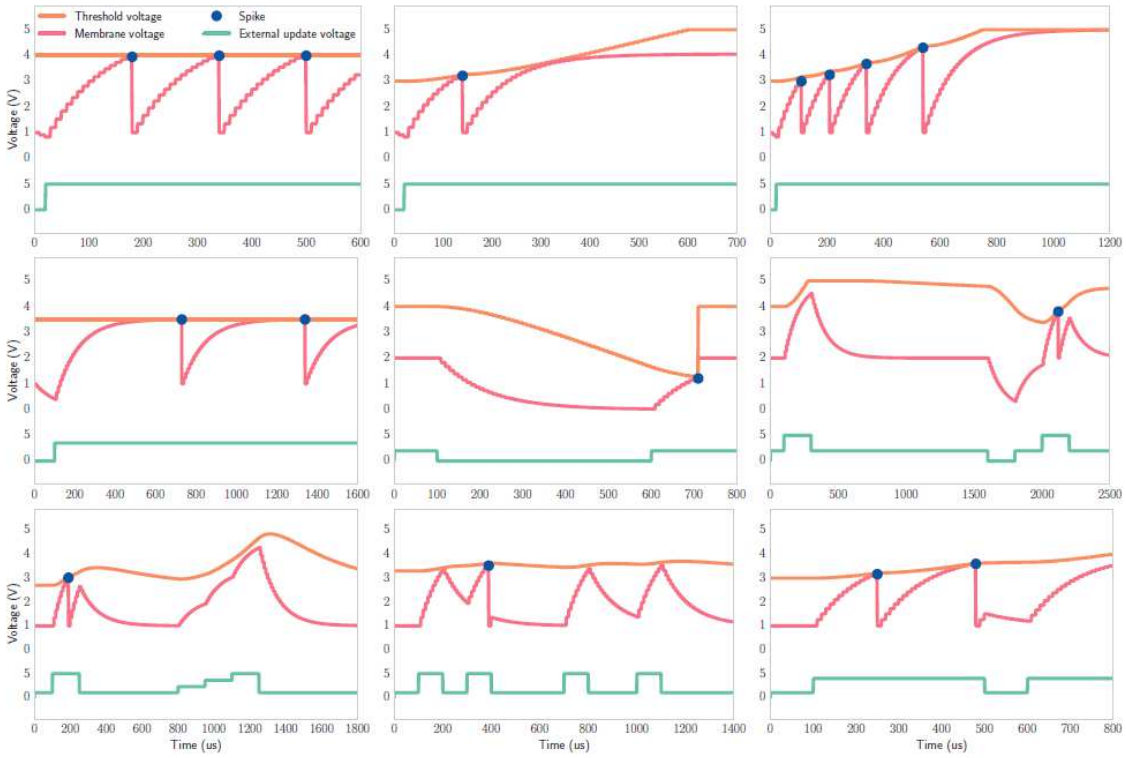


Figure 5.15: Simulation results of python model of neuron for various spiking behaviors. In order from left-to-right and top-to-bottom: Tonic Spiking, Phasic Spiking, Spike Frequency Adaptation, Class 1, Rebound Spiking, Threshold Variability, Accommodation, Integrator, Input Bistability. Note for all plots, x-axis represents time in μs and y-axis represents voltage in volts (V). Image from [30].

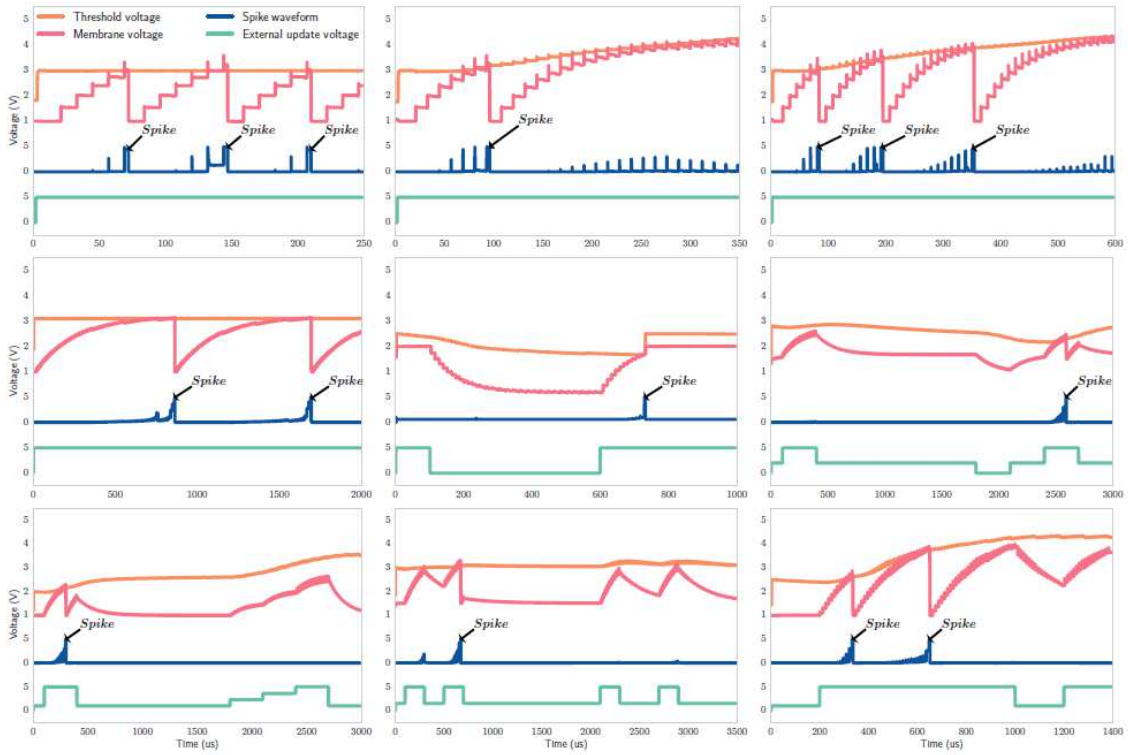


Figure 5.16: Simulation results of SPICE model of neuron for various spiking behaviors. In order from left-to-right and top-to- bottom: Tonic Spiking, Phasic Spiking, Spike Frequency Adaptation, Class 1, Rebound Spiking, Threshold Variability, Accommodation, Integrator, Input Bistability. Note for all plots, x-axis represents time in μs and y-axis represents voltage in volts (V). Image from [30].

neurons/mm^2 and [155] achieves only 387.1 neurons/mm^2. We seek to further increase the number of neurons/mm^2 by optimizing the layout of the neuron cell and implementing in smaller feature-size technology.

5.6.4 Mismatch

For analyzing mismatch (due to process variations) across the neuron array, we observed the output event to input event ratio for a fixed synaptic weight and input event rate of 1 MHz for each neuron in the array. With this fixed synaptic weight, the 2040 M-N neurons have a mean output to input event ratio of 0.0208 ±1.22e-5. In the second mode of operation, the 4080 I&F neurons have a mean output to input event ratio of 0.0222 ±5.57e-5. For fair comparison, we compare to the results from a similar experiment performed in the 0.5μm conductance-based IFAT in [77] (See Table 5.4). Our design shows significantly less deviation. Small amounts of mismatch can be taken advantage of in applications that require stochasticity. However, for those spike-based applications that do not benefit from mismatch, in this neural array, it is more controlled. This again is a result of utilizing a single, shared synapse, comparator, and threshold adaptive element for all neurons in the array. The mismatch between neurons is only due to the devices within the neuron cell itself.

[1]Characterization using I&F neurons

Table 5.4: Array Mismatch: Output Events / Input Event

Neural Array	**Mean Ratio (μ)**	**SD (σ)**	**# Neurons**
This Work[1]	0.0222	±5.57e-5	4080
Vogelstein et. al [77]	0.0210	±1.70e-3	2400

5.6.5 Array Characterization

5.6.5.1 Spiking Behaviors

The membrane voltage of the selected neuron can be observed via the oscilloscope (yellow). We also observe the threshold voltage (pink) and output spike (purple). Fig. 5.17 depicts results of an I&F neuron receiving excitatory events and spiking. Fig. 5.18 depicts a Mihalas-Niebur neuron spiking and the threshold adapting as it receives excitatory events. Fig. 5.19 depicts the effect of sending excitatory events followed by inhibitory events (removal of charge) due to a inhibitory synaptic connection. These spiking behaviors depict the ability of the neuron to operate in I&F mode, adaptive mode, and receive both inhibitory and excitatory events.

5.6.6 Power Consumption

Another design goal was to minimize power consumption. At an input event rate of 1 MHz, we measure an average power consumption of 360 μW at 5.0V power supply. A better representation of the power consumption is energy per incoming event.

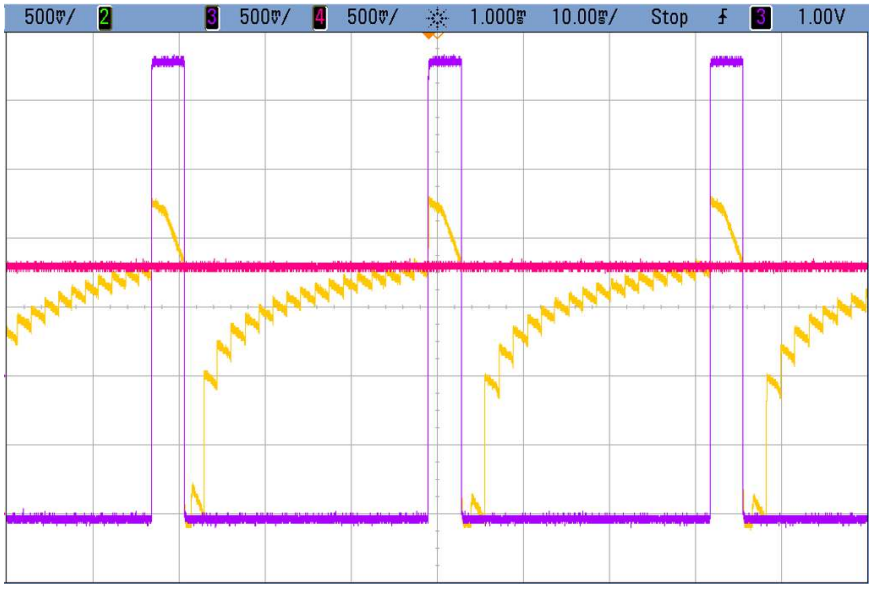


Figure 5.17: I&F neuron receiving excitatory events (no adaptation).

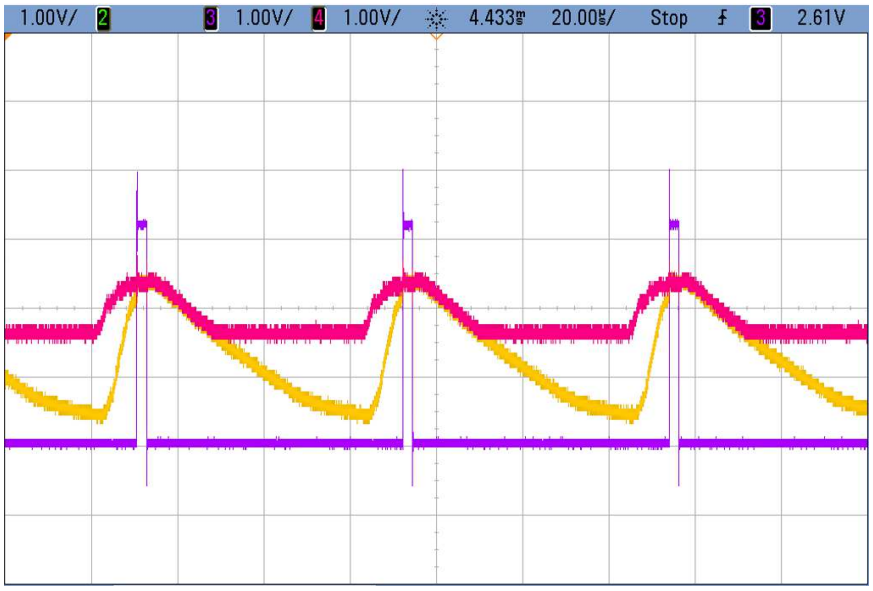


Figure 5.18: Mihalas-Niebur neuron receiving excitatory events with threshold adaptation.

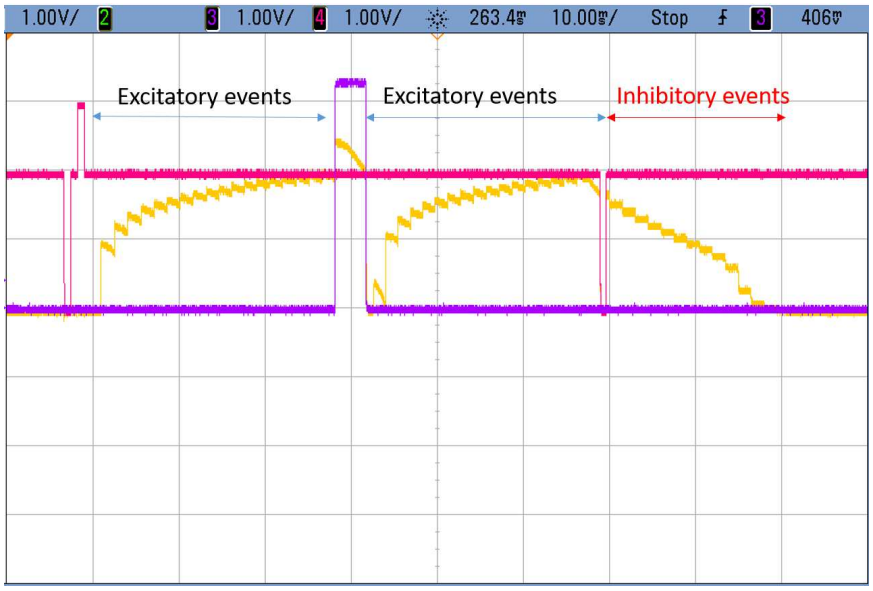


Figure 5.19: I&F neuron receiving excitatory events (no adaptation) followed by inhibitory events (removal of charge).

From these measurements, this chip consumes 360pJ of energy per synaptic event. A comparison to other state-of-the-art neural array chips can be seen in Table 5.5. Compared to those chips designed in 500nm technology [77] and 800nm [155], we see a significant reduction in energy per synaptic event. Due to complications in the circuit board, we were unable to measure low-voltage operation. However, from simulations we have validated proper operation at 1.0V (at slower speeds). Assuming dynamic energy scales with V^2 (capacitance remains the same), we estimate $\sim$14.4pJ of energy per synaptic event at 1.0V. These results are promising as these specifications will be even further optimized as we design in smaller feature-size technology.

5.6.7 Timing

As previously noted, the maximum event rate is limited by the time required to access the LUT and then assert the necessary signals for sending an event. For this model, we are further limited by the time required to reset a neuron prior to sending the next event. This is due to the fact that a neuron only outputs an event when it receives an event. Therefore, a check for an output spike must occur when an event is sent. The total required clock cycles for sending an event is 93 clock cycles. This results in a maximum event rate of $\sim$1.07 MHz at a 100 MHz FPGA clock.

Table 5.5: Neural Array Chip Comparison

Neural Array	Process	Vdd Supply	Neuron Design	Neuron Area	Energy/Event
This Work	500nm	1.0V[*2] - 5.0V	Analog	$1495\mu m^2$	14.4pJ[*3] - 360pJ
Vogelstein et. al [77]	500nm	5.0V	Analog	$2400\mu m^2$	645pJ
Indiveri et. al [155]	800nm	3.3V[*4]	Analog	$2583\mu m^2$	900pJ[*3]
Neurogrid [12]	180nm	1.8V	Analog	$1800\mu m^2$	31.2pJ
TrueNorth [27]	45nm SOI	0.85V	Digital	$3325\mu m^2$	45pJ
SpiNNaker [28]	130nm	1.2V	Digital	N/A	43nJ
BrainScales [26]	180nm	N/R[*5]	Analog	$1500\mu m^2$	N/R
Park et. al [78]	90nm	1.2V	Analog	$140\mu m^2$	22pJ
Qiao et. al [134]	180nm	N/R	Analog	$918\mu m^2$	4mW[*6]

[2]SPICE Simulation showed proper operation at 1.0V, but slower speeds
[3]Using simulated operation at 1.0V Vdd and assuming energy scales with V^2
[4]Reported Vdd and power consumption from version of neural array designed in 350nm technology
[5]Not Reported
[6]Reported average power consumption for a typical experiment

5.6.7.1 Image Processing Application

To demonstrate an application using all 4080 I&F neurons, we use the IFAT to perform an image processing application. Given an input image we generate probabilistic AE streams (on PC) such that each address corresponds to a pixel in the image (64×60). The output event-rate of each pixel has a mean firing rate that is proportional to the pixel intensity. These AEs go through a LUT implemented in memory on an FPGA, holding the associated destination addresses and synaptic weights. We first show results from a one-to-one connection between the pixel and its corresponding neuron. Secondly, we show results from a simultaneous warping (by 45°) and spreading (low-pass/blurring filter) operation in which each incoming address event is projected not only to its warped destination, but also to its neighboring neurons at that location with a synaptic weight (stored in the LUT) based on the kernel: [0.2, 0.2, 0.2, 0.2, 0.2]. The output event rates are decoded into pixel intensities and the results are shown (See Fig. 5.20). These results confirm proper operation of each neuron in the array and ability to use the complete array for performing event-based visual tasks.

5.7 Conclusion

We have demonstrated a $3mm \times 3mm$ neural array of Mihalas-Niebur neurons implemented in 500nm CMOS technology. Its novel design enables minimal power

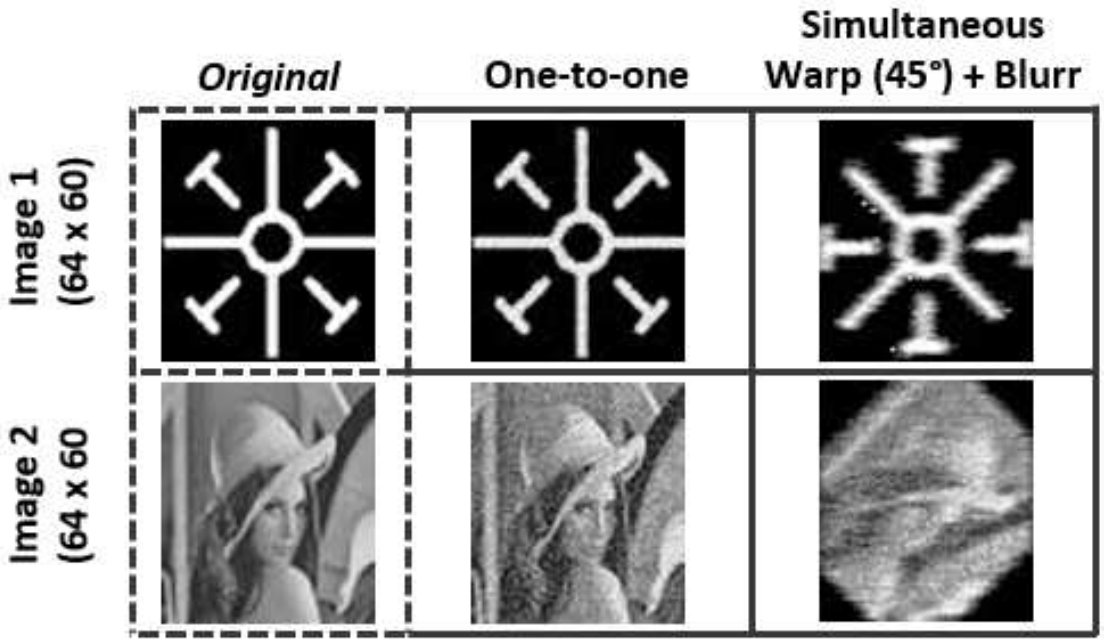


Figure 5.20: Visual processing task via chip (one-to-one and warp + blur). Image from [29].

consumption, neuron area, and mismatch by trading-off speed. However, this chip is still capable of surpassing biological real-time speeds. Speed can be increased linearly with the number of shared synapse, threshold adaptation element, and soma circuits per neural array. In continuing work we will design this architecture in a 130nm CMOS technology or smaller to further optimize neuron density and minimize power consumption. We expect neuron density of $\sim 30,000$ neurons per mm^2 in the 130nm technology. Finally, this is a generic system which can be programmed to achieve any configurability as desired by the user. This work has been published and is in review for ISCAS (International Symposium for Circuits and Systems) 2017 in Baltimore, MD.

Chapter 6

VLSI System: IFAT with Automated AER

6.1 Overview

In this chapter, we discuss a 12 × 12 IFAT designed in 55nm technology. The novelty in this array is its design in 55nm technology for high-density and low-power along with an automated AER receiver and transmitter design. We will discuss the neuron array design along with the automation software/design of the AER receiver and transmitter. This automation design is not only beneficial for the IFAT system, but also for any event-based system which takes advantage of the AER communication protocol. Finally, we will discuss results and demonstrate visual processing tasks utilizing this array of 144 I&F neurons.

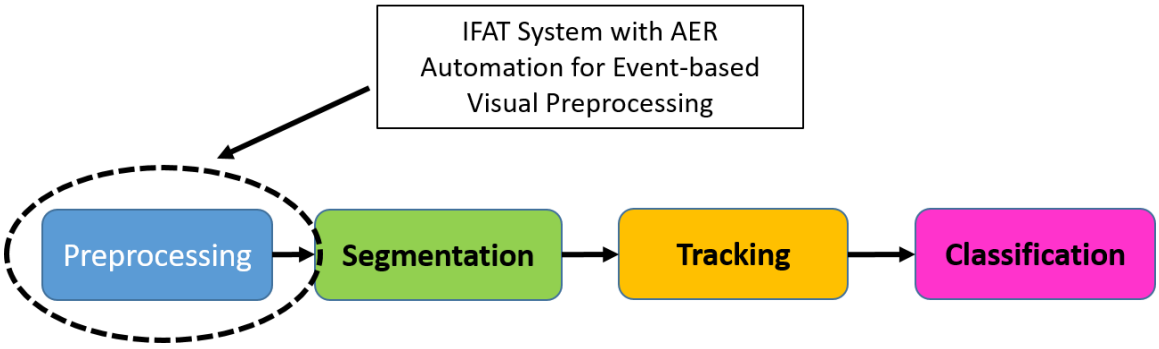


Figure 6.1: UPSIDE visual processing system flow. In this work, we focus on the pre-processing stage using the IFAT designed in 55nm.

This research was conducted under the DARPA Unconventional Processing of Signals for Intelligent Data Exploitation (UPSIDE) grant. The objective of this project was to design a visual processing system capable of object tracking (i.e. vehicles, people, etc.) on visual stimuli on the order or 2-3 gigapixels per frame. Event-based processing is utilized to achieve fast speed, high-accuracy with robustness to noise, and minimal energy consumption. This project consisted of various stages of processing (See Fig. 6.1). The purpose of this IFAT system was to fulfill the requirements and tasks within the pre-processing stage. This includes utilizing the IFAT system designed in 55nm for dewarping and filtering tasks (i.e. a debayering task). The design of the system and AER automation will be discussed in this chapter.

6.2 Related Work

This work is inspired by the VLSI-based IFAT (discussed in Chapter 5). In this work, the Mihalas-Niebur neuron model is not used, but instead the I&F neuron model is used with a conductance-based synapse. This model is similar to that used in the work of Vogelstein et al. [77]. However, the IFAT designed by [77] was designed in a 500nm process and consisted of a slightly different neuron design. In this work, a similar conductance-based synapse is used, however, the event generation circuit within the neuron is modified. Furthermore, our design in 55nm technology allows for higher density of neurons per mm^2.

Currently, most AER links use what is called bundled-data (BD) [13,64,156,157]. Bundled data incurs delay and area penalties when interfaced with delay-insensitive (DI) circuits on-chip. In BD, a four-phase handshake is used. First the request signal is asserted, then the acknowledge signal is asserted, and then the request is reset, and then the acknowledge signal is reset. In this case, the request signal must have a longer delay than the data lines because the receiver must have enough time to latch the data when it receives the request. There exists also word-serial AER which transmits and receives row and column addresses sequentially on the same data line. This lowers pad count and also improves throughput. However, the handshaking timing issues of the bundled-data delay still exist.

In this design, we utilize delay-insensitive protocol both on- and off- chip which solves these delay issues at the cost of doubling the data lines [68]. This is solved

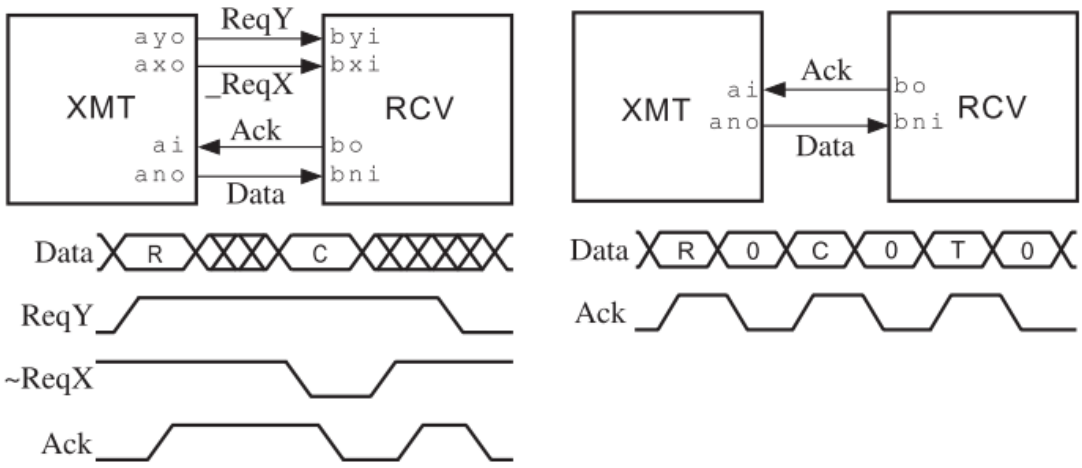


Figure 6.2: Comparison of Timing Diagram between using Bundled-Data (left) and Delay-Insensitive (right) Address Event Links

by using one-hot encoding. One-hot encoding allows representation of valid data only when a single bit is high at any given time. More specifically, 1-of-4 one-hot encoding is used for every two bits (See Table 6.1). In this case, data is valid only when a single bit is high. Henceforth, a request signal is unnecessary for signaling when valid data is ready. Furthermore, the data is sent serially such that row and column data is sent on the same data lines. There also exists a tail word that is sent following the row and column addresses which signals end of a transmission. This makes the second request signal unnecessary as well. This 1-of-4 encoding requires doubling of data lines but removes the need for request signals. An acknowledge signal is still necessary to acknowledge transmission of row, column(s), and tailword (See Fig. 6.2). First the row address is transmitted, followed by a burst of all of the

column addresses also transmitting/receiving an event at that same row. This allows for improved throughput. The final word transmitted is the tailword which signals an end of a transmission burst.

Table 6.1: 1-of-4 One Hot Encoding

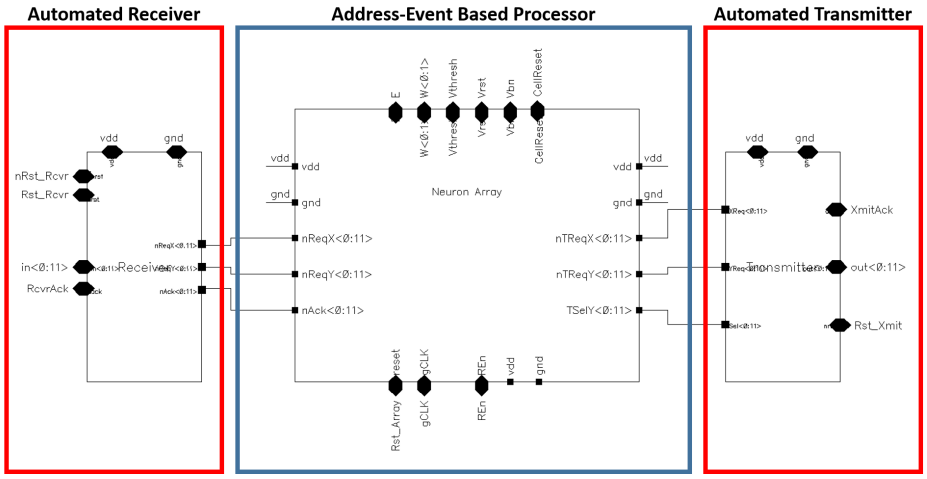
Binary	00	01	10	11
1-of-4	0001	0010	0100	1000

In this work we utilize this delay-insensitive AER design (details in [68]) and develop software for automating the AER receiver and transmitter design in the CMOS55LPX 55nm process for any neuron pitch and number of rows and columns in the array.

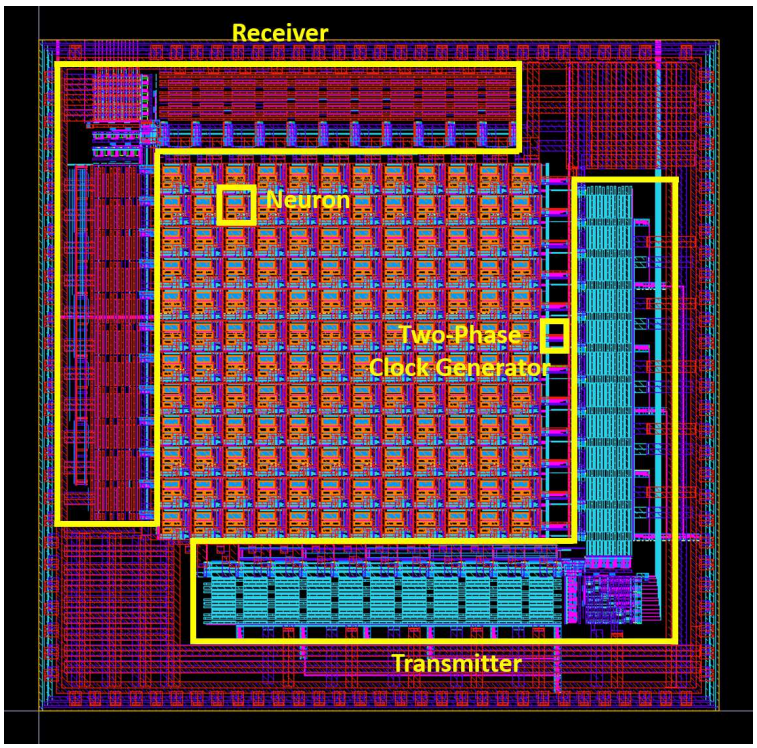
6.3 Array Design

In this work we have designed an array of 144 I&F neurons (12×12), implemented in 55nm technology. The complete schematic and layout of the neuron array can be seen in Fig. 6.3. The dimensions of the complete system is $743\mu m \times 731.6\mu m$. The neuron array consumes an area of $416\mu m \times 412\mu m$. The system consists of three primary components: the neuron, two-phase non-overlapping clock, and the AER architecture (receiver and transmitter).

The neuron design is based on the I&F neuron with a conductance-based synapse



(a) Schematic



(b) Layout

Figure 6.3: Block diagram and layout (743 µm× 731.6 µm) of complete system. The receiver and transmitter are completely automated. The user's event-based processing element array interfaces with the receiver and transmitter.

seen in [77]. As neurons receive address-events (via the receiver), charge is integrated onto the membrane capacitance. When membrane voltage (voltage across membrane capacitance), reaches a globally set threshold voltage, the neuron generates an output address-event which is outputted by the transmitter. Each neuron also contains supporting circuitry for managing the handshaking protocol between the receiver as well as with the transmitter. The neuron also contains additional circuitry for allowing bypassing of the neuron, such that an incoming address-event immediately generates an outgoing address-event at the same location. This was used for debugging purposes. The neuron cell will be discussed in Section 6.4.

The two-phase, non-overlapping clock is necessary for generating ϕ_1 and ϕ_2 pulses for the conductance-based synapse. A switch-capacitor circuit is used for this synapse within each neuron. Therefore, for each row of the neuron array, an independent two-phase clock generator is used.

Finally, the AER receiver and transmitter is responsible for handling incoming events and outgoing events. The delay-insensitive AER architecture used is that from Lin et al. [68]. It is the state-of-the-art AER architecture which uses one-hot encoding for representing address-events and controlling the handshaking protocol. The receiver can be seen on the left/top of the array and the transmitter can be seen on the right/bottom of the array. Novel software was developed via a hybrid MATLAB and SKILL script approach. The user can enter the row pitch, column pitch, number of rows, and number of columns, and the AER receiver and transmitter schematic and

layout and simulation schematics are automatically generated within a few seconds. This automated AER will be discussed in the Section 6.5.

6.4 Neuron Design

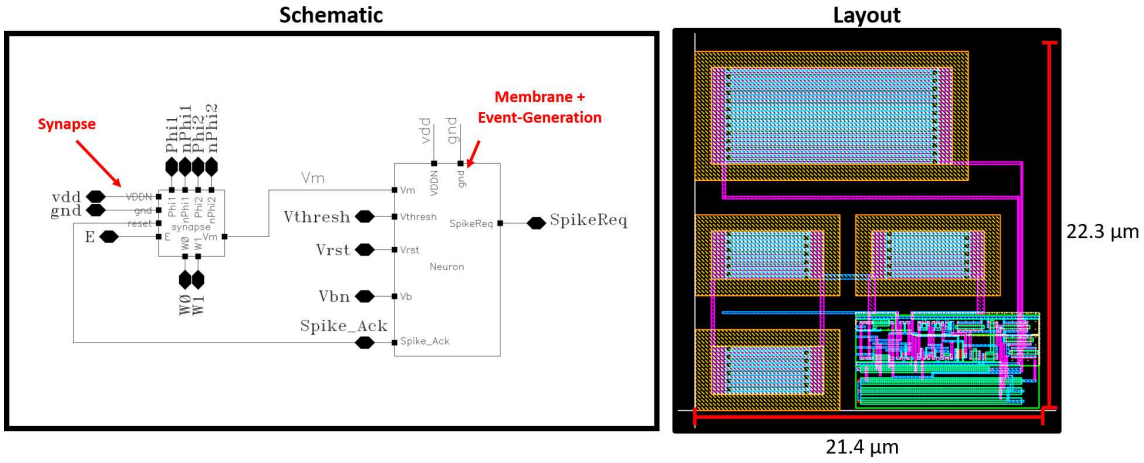


Figure 6.4: Schematic and layout of single I&F neuron. The layout of the neuron has dimensions $21.4\mu m \times 22.3\mu m$.

As previously noted, the neuron design is the I&F neuron with a conductance-based synapse inspired by Vogelstein et al. [77]. The schematic and layout of the neuron itself can be seen in Fig. 6.4. The neuron consists of two parts, the membrane capacitance with event-generation circuitry (e.g. comparator) and the synapse implemented as a switch-capacitor circuit. The complete layout dimensions of the neuron is $21.4\mu m \times 22.3\mu m$.

6.4.1 Membrane and Event-Generation Circuit

The membrane capacitance and event generation circuit schematic can be seen in Fig. 6.5.

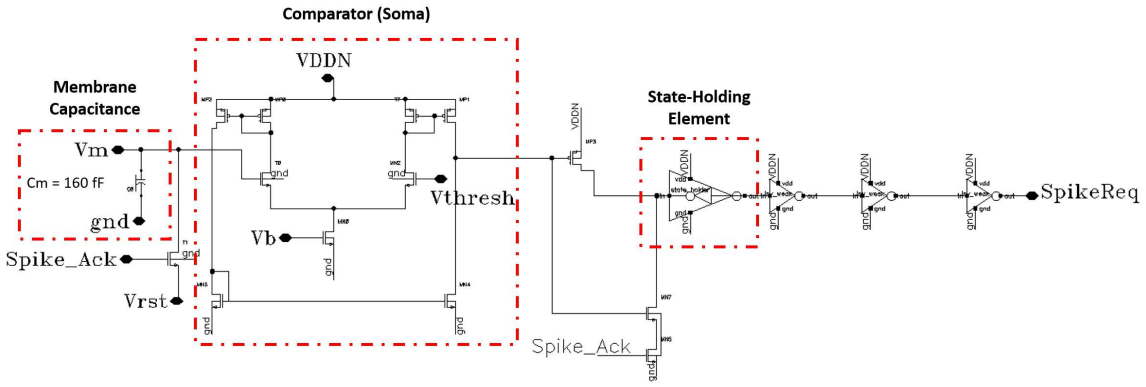


Figure 6.5: Schematic of Membrane and Event-Generation Circuit.

The membrane capacitance is 160 fF and uses vertical natural capacitors (vn caps). When the voltage across the capacitance exceeds the threshold voltage, the comparator will output an logic low signal (GND). This logic low signal is the input to a PMOS transistor with its drain connected to the input of a state holding element. Three series inverters are connected to the output of the state holding element. The output of these inverters is the *Spike_Req* signal which outputs a logic high when an event is generated. When an acknowledge signal is received (*Spike_Ack* = "1"), the voltage across the membrane capacitance is reset to $Vrst$ (usually GND). This caused the comparator output to go high, which in turn causes the input to the state holding machine to be pulled low. In result, *Spike_Req* also gets pulled low.

6.4.2 Synapse Circuit

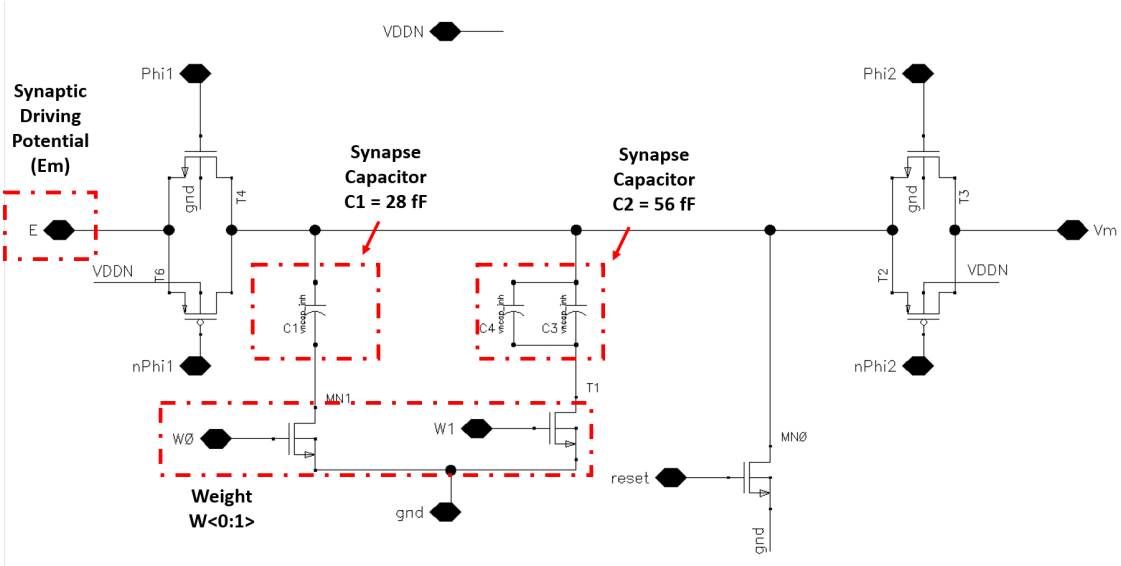


Figure 6.6: Schematic Neuron Synapse Circuit.

The synapse circuit is implemented as a switch-capacitor circuit (Fig. 6.6). A ϕ_1 and ϕ_2 pulse is applied when a neuron receives an event. This deposits a charge onto the membrane capacitance proportional to $Em - Vm$ and $W < 0:1 >$. The weights $W < 0 >$ and $W < 1 >$ activate switches which determine the size of the synapse capacitance. The larger the synapse capacitance, the more charge that is integrated onto the membrane capacitance. $W < 0 >$ controls the a synapse capacitor of 28 fF ($C1$) while $W < 1 >$ controls the synapse capacitor of size 56 fF ($C2 = 2 \times C1$). A reset signal (tied to *Spike_Ack*) resets the charge on the synapse capacitance when the *Spike_Ack* signal is asserted.

6.4.3 Neuron Cell Supporting Circuitry

Within each of the complete neuron cells in the array, there is supporting circuitry for controlling the receiving of events, handling of the input two-phase clock, and transmission of events generated by the neuron circuit. The complete schematic diagram and layout of the complete neuron cell including the neuron circuit ("IAF Full Neuron") and its supporting digital blocks can be seen in Fig. 6.7 and Fig. 6.8. Each complete neuron cell (including digital control) is 34.6μm in width and 34.2μm in height.

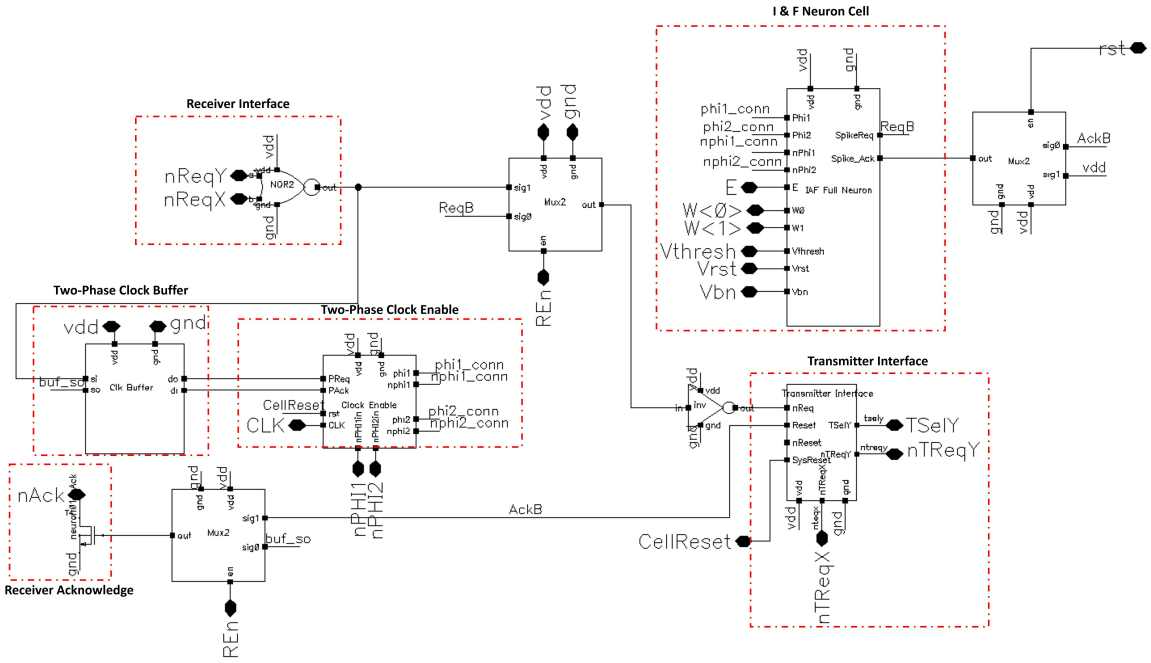


Figure 6.7: Schematic of Complete Neuron Cell with Supporting Circuitry.

For controlling the receiving of events, the sequence begins at the inputs $nReqY$ and $nReqX$. These are inputs from the receiver block. When both of these signals

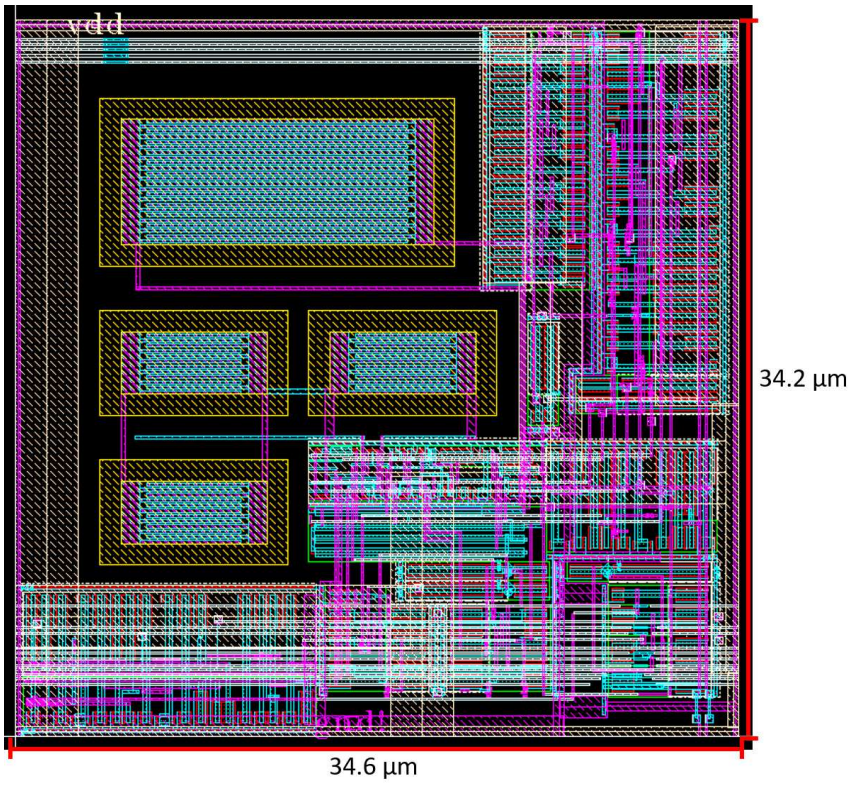


Figure 6.8: Layout of Complete Neuron Cell with Supporting Circuitry ($34.6\mu m \times 34.2\mu m$).

go low, the NOR gate output goes high. There a various 2-input multiplexers (implemented using three NAND gates in a tree-like fashion) used in the neuron. These allow for two possible modes of the neuron array. The first mode is when input REn is high (and inverted input $nREn$ is low). In this mode, the IFAT neuron circuit is bypassed and when events are received, they are immediately transmitted. The second mode is when REn is low (and $nREn$ is high). In this mode, the IFAT neuron circuit is used and when an event is received, it is relayed to the neuron circuit. So, once an event is received and $nReqY$ and $nReqX$ are both low, if in the first mode where the neuron is bypassed, the event is immediately directed to the transmitter interface. However, in the second mode, the output is coupled to the clock buffer ("Clk Buffer"), which ensures a two-phase clock has been outputted and processed before processing another event. When the output of the clock buffer, *so* goes high, it turns on the NMOS transistor which pulls $nAck$ low (coupled to the receiver). When *do* of the clock buffer goes high, it signals the clock enable block to generate a single $Phi1$ and $Phi2$ non-overlapping clock pulse to the neuron circuit. Finally, when the neuron reaches threshold and transmits an output spike/event request, it signal the Transmitter interface to output a row event request via the $nTReqY$. The acknowledge from the transmitter is tied to $nTSelY$. When the acknowledge signal goes high, $nTReqX$ is pulled low which signals the column request. When the $nTSelY$ acknowledge goes low, the neuron has been reset, signaling the IFAT neuron circuit $Spike_{Ack}$ signal (logic high), resetting the neuron.

6.5 Automated AER Design

6.5.1 User-End

The key component of this system is its simplicity from the user's perspective. All SKILL scripts are generated using MATLAB. The user simply executes the top-level MATLAB function which generates the final SKILL scripts for the transmitter and receiver. This MATLAB top-level function takes in the following four inputs:

1. Number of Rows (even only)
2. Number of Columns (even only)
3. Row Pitch (in microns)
4. Column Pitch (in microns)

Within seconds, the SKILL scripts for the AER Receiver and AER Transmitter are automatically created independently. The user then simply runs each SKILL script independently in Cadence and within seconds the complete layout, schematic, and symbol of the receiver and transmitter or created. The entire receiver and transmitter layout will LVS and DRC correctly for any number of even rows and columns. It will also generate schematics for simulating the receiver and transmitter. The user simply must ensure the processing elements in their array contains the correct signals for communicating with the receiver and transmitter (See Fig. 6.3).

6.5.2 Design Approach

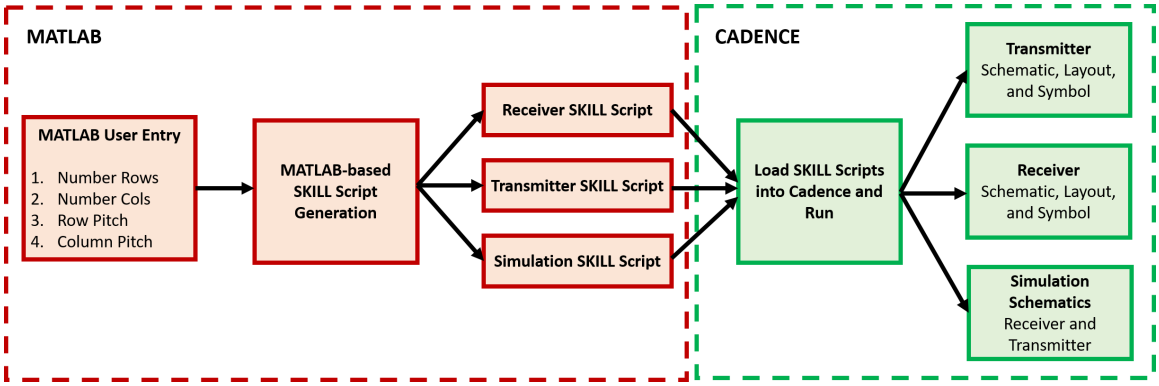
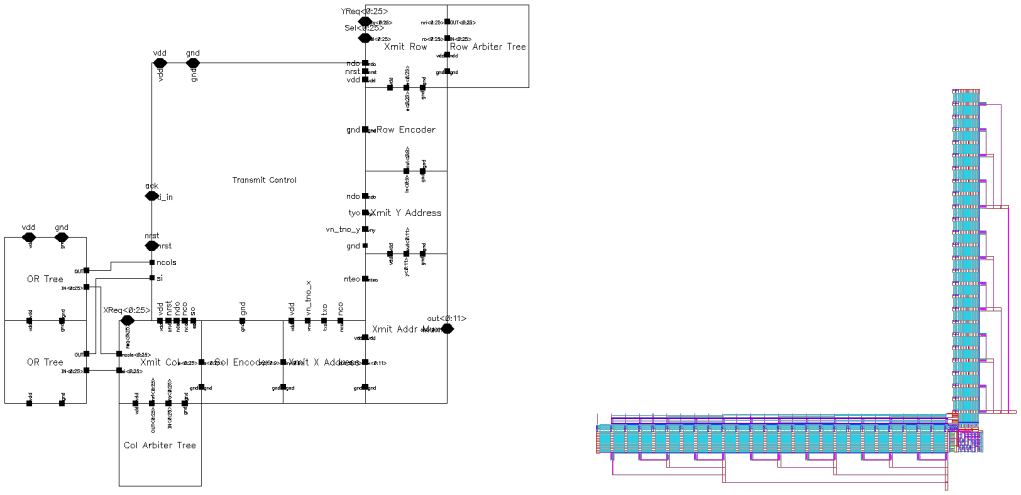


Figure 6.9: AER Automation Flow. MATLAB is used for generating SKILL scripts for the transmitter, receiver, and simulation schematics. These SKILL scripts are loaded into Virtuoso/Cadence and run, generating the LVS and DRC-free schematic, layout, and symbol for the receiver and transmitter.

This approach to the AER automation consisted of three major components. The first is the Cadence library of 100+ cells that are used for automating both the transmitter and receiver. This library is called *AER Automation*. The second component is the SKILL script. The automation generates two SKILL scripts, one called *create_receiver.il* for the receiver and the second called *create_transmitter.il* for the transmitter. These SKILL scripts use SKILL language for calling functions which essentially place appropriate cells, wires, pins, symbols, etc. in their appropriate positions for layouts, schematics, and symbols. The final component is MATLAB which is the key source of the automation. MATLAB functions were written and are responsible for performing various computations for proper placement and naming of

cells, wires, pins, etc. depending on the number of rows/columns and row/column pitch entered by the user. There is quite a bit of computation that must be performed and MATLAB is ideal for such computation. There is a MATLAB function (*makeAERParams.m*) which creates a structure for holding all of the parameters for every cell used in the automation process (cell name, cell size, metal layers for various signals, positioning, etc.). Finally, there are MATLAB functions for creating the layout, schematic, and symbol for each of the major blocks for both the transmitter and receiver. Each of these functions accepts the fileID for writing to either the transmitter SKILL file or the receiver SKILL file. One of these SKILL functions connects all of the generated major blocks appropriately (for both receiver and transmitter). Therefore, after running the top-level MATLAB function, each of the two SKILL scripts will contain multiple SKILL functions for creating layout, schematic, and symbol of individual blocks, and then a final SKILL function calls all of these functions. All of this occurs within a single SKILL file (for both receiver and transmitter). This allows the user to simply run the top-level MATLAB function, then load the generated *create_receiver.il* and run *create_all*() function and load *create_transmitter.il* and run *create_all*() in the Cadence Virtuoso window. Then, the complete transmitter and receiver blocks are generated (passing LVS and DRC checks). The flow of the AER Automation can be seen in Fig. 6.9.

6.6 Transmitter Automation



(a) Schematic (b) Layout

Figure 6.10: Schematic and layout of the complete delay-insensitive transmitter.

The complete transmitter (layout, schematic, and symbol) is automatically generated given the number of rows (even only), number of columns (even only), and the row and column pitch of the user's event-based processing element array. The transmitter automation is a piece-wise process. It involves first the automation of the "major blocks" involved in the transmitter design. Finally, it involves the proper arrangement/connection of these blocks to create the complete functioning transmitter. All of the subcells used for generating these major blocks are located in the *AER Automation* library. As previously noted, MATLAB is used for holding all of the necessary parameters, performing various computations for generating accurate SKILL functions that create all of the major blocks and arrangement of these blocks

to form the complete transmitter in schematic and layout.

In the proceeding sections, we will discuss the how each of the major blocks involved in the automation process are generated. The complete transmitter schematic and layout can be seen in Fig. 6.10. It is composed of the major blocks (including filler cells) that were automated and then interfaced for generating the complete transmitter. These blocks are listed in Table 6.2.

In regards to the transmitter section of this report, first, the signals and timing of the signals required for interfacing with the transmitter will be discussed. Then, constraints/specifications and simulations results will be discussed. Finally, the automation of each of the major blocks above along with the generation of the complete transmitter will be discussed.

6.6.1 Interfacing Transmitter with an Address-Event-Based Processor Array

The transmitter automation is independent of the user's address-event-based processor array. However, the communication protocol of the transmitter must be considered when designing the interface between the user's processor array and the transmitter. Each processor element in the array must contain specific signals and the control of these signals must operate in a specific way for proper communication with

Table 6.2: List of major blocks that were automated and used for the complete transmitter generation.

Transmitter Block	**Cell Name (in Cadence)**	**Description**
Transmit Row [Address]	$XmitRow$	Latches the selected row from the array.
Row Encoder	$rowEncoder$	Encodes the selected row into 1-of-4 encoding.
Transmit Y Address [Final]	$XmitYAddr$	Latches the final encoded row address to be outputted.
Transmit Column [Address]	$XmitCol$	Latches the selected column from the array.
Column Encoder	$colEncoder$	Encodes the selected column into 1-of-4 encoding
Transmit X Address [Final]	$XmitXAddr$	Latches the final encoded column address to be outputted.
OR Tree	OR_Tree	OR Tree used for creating the OR Stack of transmitter.
OR Tree Stack	$ORTree\_Stack$	Checks if there is a column request and latched column request.
XmitYAddr and Row Encoder Bridge/Filler	$xmityaddr\_rowenc\_bridge$	Bridge between $XmitYAddr$ and $rowEncoder$.
XmitXAddr and Col Encoder Bridge/Filler	$xmitxaddr\_colenc\_bridge$	Bridge between $XmitXAddr$ and $colEncoder$.
Transmitter Control Top Bridge/Filler	$xmitcontrol\_top\_fill$	Bridge at top of $XmitControl\_Final$.
Transmit Address Mux	$XmitAddrMux$	Transmits final row address, column address, and tailword.
Row Arbiter Tree	$RowArbiter$	Selects a single row when multiple row requests.
Column Arbiter Tree	$ColArbiter$	Selects a single column when multiple row column.
Transmitter Control	$XmitControl\_Final$	Asynchronous control of the complete transmitter.

the transmitter. The block diagram of the transmitter and processor array interface can be seen in Fig. 6.11. Assuming Y number of rows and X number of columns, the signals for interfacing the transmitter with the processor array can be seen in 6.3.

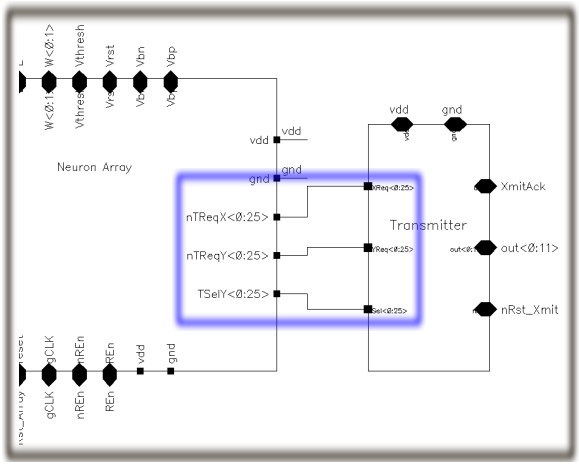


Figure 6.11: Transmitter interface with address-event-based processor array. The blue box surrounds the three signals used for communication with the transmitter.

Table 6.3: Transmitter Signals for Interfacing with Processor Array (X Columns, Y Rows)

Transmitter Signal	**Direction**	**Description**
$nTReqX < 0 : X >$	Input	Column request from processor array (active low signal)
$nTReqY < 0 : Y >$	Input	Row request from processor array (active low signal)
$TSelY < 0 : Y >$	Output	Row select from transmitter (active high signal)

Therefore, each event-based processing element in the array must have a row request output ($nTReqY$), column request output ($nTReqX$), and row select input ($TSelY$). Each row of processing elements of the array should be tied to a single $nTReqY$ wire (wired-OR). Each column of processing elements of the array should

be tied to a single $nTReqX$ wire (wired-OR). Considering the transmitter architecture is asynchronous, the user has flexibility in the control of these signals. These signals can be either synchronous or asynchronous. However, the order of assertion of these signals is important for proper communication between the transmitter and the processor array. The order of operations is as follows:

1. Processor element $nTReqY$ signal is asserted low.
2. Wait for $TSelY$ to go high (input from transmitter).
3. Processor element $nTReqX$ signal is asserted low.
4. Wait for $TSelY$ to go low (input from transmitter).
5. Signals $nTReqY$ and $nTReqX$ should be deasserted.

6.6.1.1 Processing Element to Transmitter Communication Block

To further facilitate the communication, a block was designed that can be placed within each processor element that acts as a bridge between the event-based processor element and the transmitter. This block is called $xmitinterface$ and is placed within each element. The schematic of this block can be seen in Fig. 6.12. The $nReq$ signal is asserted low when the processor element is ready to output an event. If $TSelY$ is low, the nMOS transistor with the drain connected to $nTReqY$ is turned on. This pulls $nTReqY$ low. This $nTReqY$ signal is tied to the same block within each processing element in the corresponding row (wired-OR). The transmitter will then acknowledge

this row request by pulling $TSelY$ high. This pulls $nReset$ low which then pulls $Reset$ high, which then turns on the nMOS with its drain connected to $nTReqX$. The $Reset$ signal is connected to the processing element and signals the processor that the event has been acknowledged and can be reset. Similarly to $nTReqY$, the signal $nTReqX$ is connected to the same signal within each processing element in the corresponding column (wired-OR). When the transmitter has acknowledged the row request and column request, $TSelY$ eventually goes low and which causes $nReset$ to go high and $Reset$ to go low while the nMOS transistors controlling $nTReqY$ and $nTReqX$ are turned off. This block further simplifies the use of this automated transmitter with any event-based processor array.

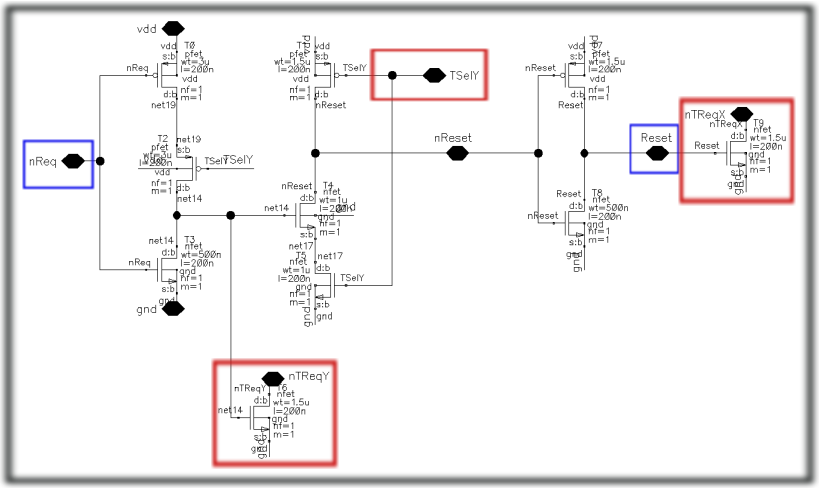


Figure 6.12: Schematic of the block placed within each processor element cell to facilitate communication between the event-based processor and the transmitter. Red: Signals connected to transmitter. Blue: signals connected to event-based processor element.

6.6.2 Transmit Row [Address] Automation

The Transmit Row [Address] interfaces directly with the row request ($nTReqY$) and row select ($TSelY$) from the processor array. It is responsible for receiving the request, outputting the request to the row arbiter, latching the selected row address, and outputting the selected row address to the row encoder. The automation consisted of placing the subcell, *rowArbLogic*, for each row (one vertical column of *rowArbLogic* cells). Extension of metal wires in between each *rowArbLogic* cell for *vdd*, *gnd*, *nrst*, and *ndo* was necessary when the vertical pitch specified by the user was larger than the height of the *rowArbLogic* cell (See Fig. 6.13).

Table 6.4: Subcells used for automation of the Transmit Row [Address].

Subcells Used	**Dimensions ($W \times H$)**	**Description**
rowArbLogic	$14.459\mu m \times 9.1\mu m$	Connected to $TSelY$ and $nTReqY$ of the processor array. Also latches row selection to row encoder input.

6.6.2.1 Block Dimensions (Layout)

Given a row pitch of $pitch_{vert}$. The $XmitRow$ block dimensions are:

$Width = 14.459\mu m$

$Height = pitch_{vert} \times num\_rows$

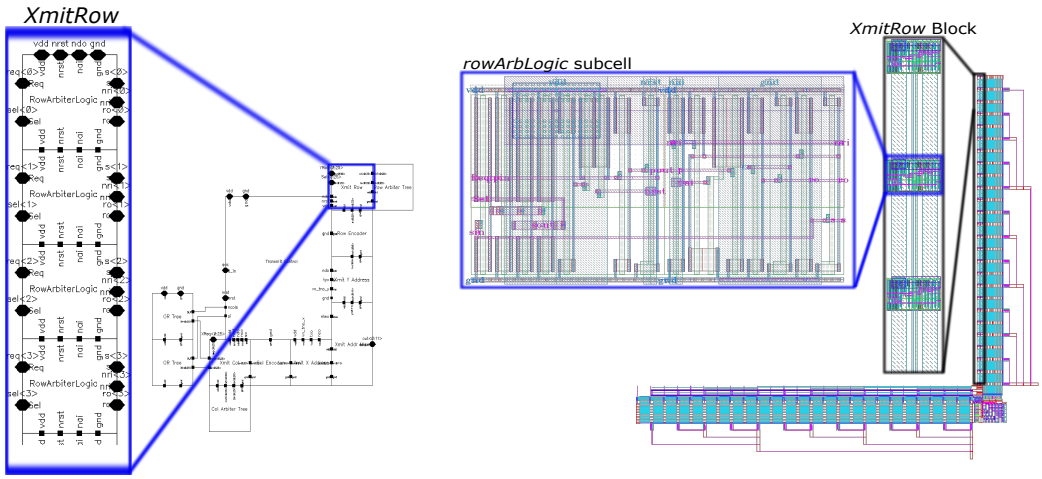
6.6.2.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

createXmitRow_layout.m : Writes a SKILL function for generating the *XmitRow* layout.

createXmitRow_schem.m : Writes a SKILL function for generating the *XmitRow* schematic.

createXmitRow_symbol.m : Writes a SKILL function for generating the *XmitRow* symbol.



(a) Schematic (b) Layout

Figure 6.13: Automated layout and schematic of the XmitRow block.

6.6.3 Row Encoder Automation

The Row Encoder is responsible for encoding the selected row address component of the address-event to a 1-of-4 one-hot encoded address. For example, if row 4 were selected, this would represent 100 in decimal. The 1-of-4 one-hot encoding would then encode this binary value into 1-of-4 encoded value. Every two bits is represented by four bits so 0100 would be encoded to 00100001 (01 $\rightarrow$ 0010 and 00 $\rightarrow$ 0001). MATLAB is responsible for computing the number of outputs required given any number of rows in the processor array. The encoder must then be automated appropriately given number of rows and number of outputs required. To compute the number of necessary 1-of-4 groups given the number of rows, the following equation is used:

$$num\_bits = ceil(log_2(num\_rows)) \tag{6.1}$$

$$num\_1of4\_groups = ceil(num\_bits/2) \tag{6.2}$$

num_bits represents the number of bits required for the binary representation of the number number of rows. *num_rows* is the number of rows in the processor array. Considering every two bits requires one 1-of-4 group, we divide *num_bit* by 2. To encode each pair of bits into its appropriate 1-of-4 group, we automate a *subencoder* for each 1-of-4 group. The number of *subencoders* is equal to the number of 1-of-4 groups. Each 1-of-4 group has 4 outputs and therefore, the number of total output

bits is:

$$num\_output\_bits = num\_1of4\_groups \times 4 \tag{6.3}$$

MATLAB is used to compute these values and then further, automate layout, schematic, and symbol of each *subencoder* (*Enc*1, *Enc*2, *Enc*3, ...). For each pair of bits, the 1-of-4 group is encoded differently and therefore, quite a bit of computation must take place for proper generation of each *subencoder* for accurate encoding into its 1-of-4 representation at the output (See Fig. 6.14). Each *subencoder* uses multiple *Enc_nmos* subcells. The arrangement of these *Enc_nmos* subcells is vital for proper functioning of each *subencoder*. Considering the same inputs feed into each *subencoder*, these *subencoders* are then connected to form the final *rowEncoder* block.

Table 6.5: Subcells used for automation of the Row Encoder.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
Enc_nmos	$4.2\mu m \times 9.1\mu m$	Single nMOS transistor block whos drain is connected to an output line, gate connected to the an input row line, and source tied to ground.
Enc_nmos_dummy	$4.2\mu m \times 9.1\mu m$	Filler block such that various input and output signals remain connected appropriately.

6.6.3.1 Block Dimensions (Layout)

Given a row pitch of $pitch_{vert}$. The *rowEncoder* block dimensions are:

$Width = 4.2\mu m \times num\_output\_bits$

$Height = pitch_{vert} \times num\_rows$

6.6.3.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL functions which automate layout, schematic, and symbol of this block:

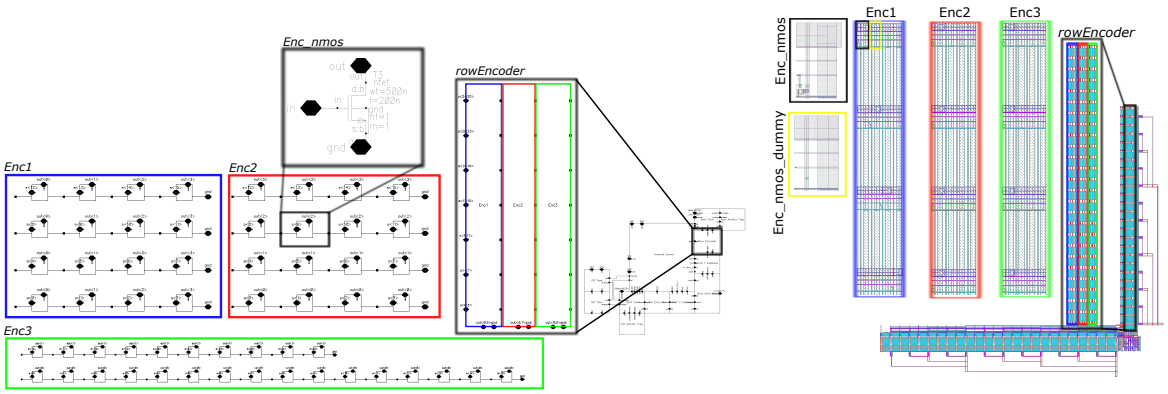
createXmitRowEncoder_layout.m : Writes a SKILL function for generating the *rowEncoder* layout.

createXmitRowEncoder_schem.m : Writes a SKILL function for generating the *rowEncoder* schematic.

createXmitRowEncoder_symbol.m : Writes a SKILL function for generating the *rowEncoder* symbol.

6.6.4 Transmit Y Address [Final] Automation

The Transmit Y Address [Final] block is responsible for latching the encoded row address to the Transmit Address Mux block such that it can be outputted from the transmitter. This latching is controlled by various signals from the Transmitter Control block. The Transmit Y Address [Final] block is composed of arrangement of



(a) Schematic (b) Layout

Figure 6.14: Automated layout and schematic of the rowEncoder block. The subencoders Enc1, Enc2, and Enc3 are only partially shown.

three different subcells. The first is *addrlatchY* which is directly coupled with the control signals *ndo* and *tyo* from the Transmitter Control block. Each *addrlatchY* block receives two input bits (from a pair of output bits from the rowEncoder output). When signaled by the Transmitter Control, these input bits are latched to the output of the *addrlatchY* cell. However, it is important to note that depending on the number of rows in the processor array, not every output bit/line from the rowEncoder may be dependent on the input row requests. Therefore, for these remaining bits, they are tied to Vdd. So, there exists $addrlatchY\_VDDB$ cell, which is used when both inputs should be tied to Vdd. There also exists an $addrlatchY\_VDDR$ cell, which is used when only the right (MSB) of the pair of bits should be tied to Vdd (See Fig. 6.15). The total number of *addrlatchY* blocks that should be placed side-by-side

is:

$$num\_addrlatchy\_blocks = num\_output\_bits/2 \tag{6.4}$$

The second subcell used is $validneutral1in4$. This takes in two pairs of output bits from two $addrlatchY$ cells (total of 4 inputs). This subcell is responsible for checking that only one of the 4 inputs are high (valid 1-of-4 code). The total number of $validneutral1in4$ blocks placed side-by-side is:

$$num\_validneutral\_blocks = num\_addrlatchy\_blocks/2 \tag{6.5}$$

The output of each $validneutral1in4$ block is fed into a tree of C-element subcells. The C-element cells output logic high when both inputs are logic high and do not change state until both inputs go logic low (using state-holding elements). There are two different subcells for the C-elements, $Celem1$ and $Celem2$. The $Celem1$ block is used mostly however, on the first level of the tree, if it is an odd number of input then $Celem1$ is used. To conserve area, the C-element blocks are placed side-by-side even though they are connected in a tree-structure. There is automation for drawing metal wires which connect the C-elements in the tree-like structure (See Fig. 6.15). The output of the C-element tree is vny which is connected to the Transmitter Control block. The total number of C-element blocks required is always:

$$num\_celem\_blocks = num\_validneutral\_blocks - 1 \tag{6.6}$$

The complete arrangement of these subcells in schematic and layout can be seen in Fig. 6.15.

Table 6.6: Subcells used for automation of the Transmit Y Address [Final].

Subcells Used	**Dimensions ($W \times H$)**	**Description**
$addrLatchY$	$8.4\mu m \times 12.5\mu m$	Input is single pair of bits from rowEncoder output bus. Latches this rowEncoder output to output of the $XmitYAddr$ cell.
$addrLatchY\_VDDR$	$8.4\mu m \times 12.5\mu m$	Input is last bit (MSB) from rowEncoder output bus. Second input is instead tied to Vdd. Latches this rowEncoder output to output of the $XmitYAddr$ cell.
$addrLatchY\_VDDB$	$8.4\mu m \times 12.5\mu m$	Both inputs are tied to Vdd. Necessary when last two bits (MSB) are not dependent on row requests. Latches this rowEncoder output to output of the $XmitYAddr$ cell.
$validneutral1in4$	$16.8\mu m \times 5.1\mu m$	Checks that 4 inputs represent valid 1-of-4 one-hot encoded group.
$Celem1$	$16.8\mu m \times 3.4\mu m$	C-element used when odd number of inputs (odd number of $validneutral1in4$ outputs).
$Celem2$	$16.8\mu m \times 3.4\mu m$	C-element used throughout most of C-element tree-structure.

6.6.4.1 Block Dimensions (Layout)

The $XmitYAddr$ block dimensions are:

$Width = 16.8\mu m \times num\_1of4\_groups$

$Height \approx 21\mu m$

6.6.4.2 MATLAB Files

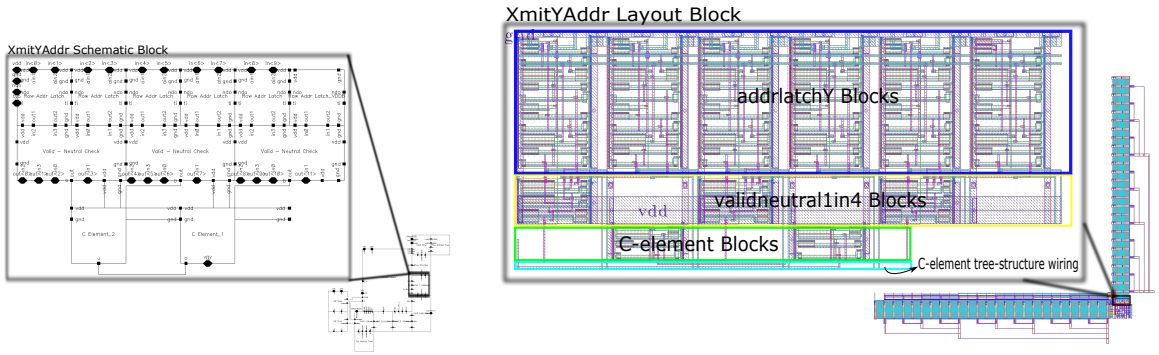
The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

$createXmitYAddr\_layout.m$: Writes a SKILL function for generating the $XmitYAddr$

layout.

createXmitYAddr_schem.m : Writes a SKILL function for generating the *XmitYAddr* schematic.

createXmitYAddr_symbol.m : Writes a SKILL function for generating the *XmitYAddr* symbol.



(a) Schematic (b) Layout

Figure 6.15: Automated layout and schematic of the XmitYAddr block.

6.6.5 Transmit Column [Address] Automation

The Transmit Column [Address] interfaces directly with the column request ($nTReqX$) from the processor array. It is responsible for receiving the request, outputting the request to the column arbiter, latching the selected column address, and outputting the selected column address to the column encoder. It is also responsible for outputting buses $ncols < 0 : X >$ and $si < 0 : X >$ (X is number of columns) to two OR-Trees

for checking that at least one column is selected. The automation consisted of placing the subcell, *colArbLogic*, for each column (one horizontal row of *colArbLogic* cells). Extension of metal wires in between each *colArbLogic* cell for *vdd*, *gnd*, *nrst*, *nco*, *so*, and *ndo* was necessary when the horizontal pitch specified by the user was larger than the height of the *colArbLogic* cell (See Fig. 6.16).

Table 6.7: Subcells used for automation of the Transmit Column [Address].

Subcells Used	**Dimensions ($W \times H$)**	**Description**
$colArbLogic$	$16.419\mu m \times 9.1\mu m$	Connected to $nTReqX$ of the processor array. Also latches column selection to column encoder input. Output of column request for that column $ncols$ is tied to an OR-Tree. Column selection si is also outputted to an OR-Tree.

6.6.5.1 Block Dimensions (Layout)

Given a column pitch of $pitch_{horiz}$. The $XmitCol$ block dimensions are:

$Width = pitch_{horiz} \times num\_cols$

$Height = 16.419\mu m$

6.6.5.2 MATLAB Files

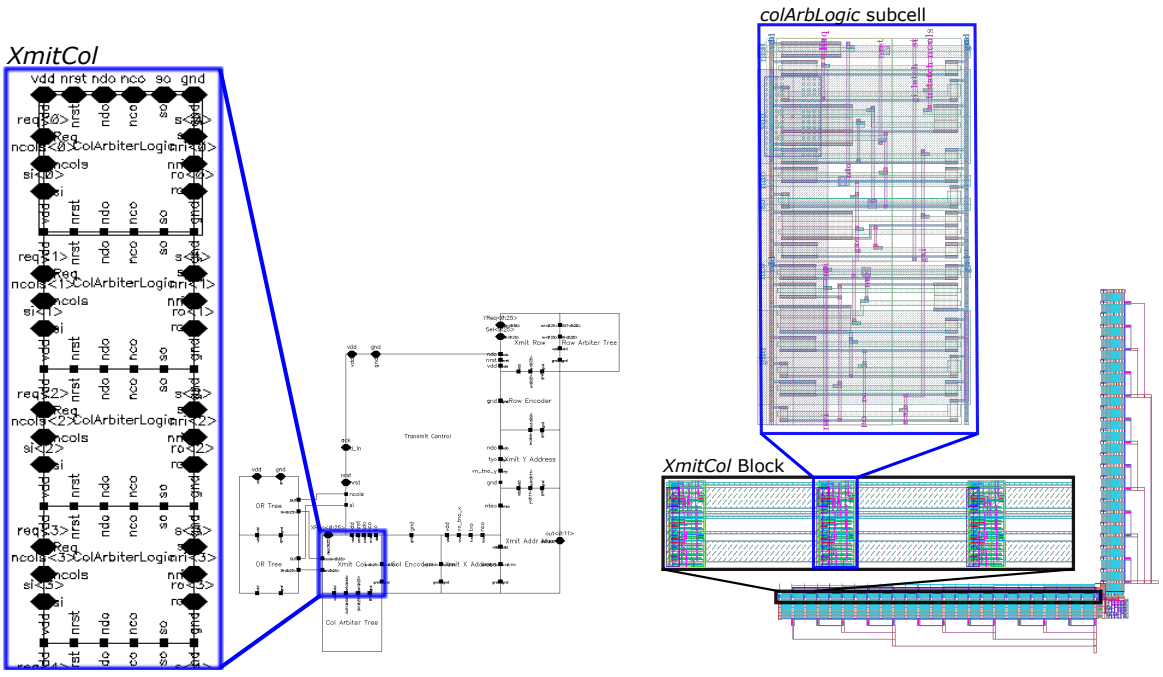
The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this

block:

createXmitCol_layout.m : Writes a SKILL function for generating the *XmitCol* layout.

createXmitCol_schem.m : Writes a SKILL function for generating the *XmitCol* schematic.

createXmitCol_symbol.m : Writes a SKILL function for generating the *XmitCol* symbol.



(a) Schematic (b) Layout

Figure 6.16: Automated layout and schematic of the XmitCol block.

6.6.6 Column Encoder Automation

The Column Encoder functions identically to the Row Encoder. It is responsible for encoding the selected column address component of the address-event to a 1-of-4 one-hot encoded address. For example, if column 4 were selected, this would represent 100 in decimal. The 1-of-4 one-hot encoding would then encode this binary value into 1-of-4 encoded value. Every two bits is represented by four bits so 0100 would be encoded to 00100001 (01 $\rightarrow$ 0010 and 00 $\rightarrow$ 0001). MATLAB is responsible for computing the number of outputs required given any number of columns in the processor array. The encoder must then be automated appropriately given number of column and number of outputs required. To compute the number of necessary 1-of-4 groups given the number of columns, the following equation is used:

$$num\_bits = ceil(log_2(num\_cols)) \tag{6.7}$$

$$num\_1of4\_groups = ceil(num\_bits/2) \tag{6.8}$$

num_bits represents the number of bits required for the binary representation of the number number of columns. *num_cols* is the number of column in the processor array. Considering every two bits requires one 1-of-4 group, we divide *num_bit* by 2. To encode each pair of bits into its appropriate 1-of-4 group, we automate a *subencoder* for each 1-of-4 group. The number of *subencoders* is equal to the number of 1-of-4 groups. Each 1-of-4 group has 4 outputs and therefore, the number of total

output bits is:

$$num\_output\_bits = num\_1of4\_groups \times 4 \tag{6.9}$$

MATLAB is used to compute these values and then further, automate layout, schematic, and symbol of each *subencoder* (*Enc*1, *Enc*2, *Enc*3, ...). For each pair of bits, the 1-of-4 group is encoded differently and therefore, quite a bit of computation must take place for proper generation of each *subencoder* for accurate encoding into its 1-of-4 representation at the output (See Fig. 6.17). Each *subencoder* uses multiple *Enc_nmos_col* subcells. The arrangement of these *Enc_nmos_col* subcells is vital for proper functioning of each *subencoder*. Considering the same inputs feed into each *subencoder*, these *subencoders* are then connected to form the final *colEncoder* block.

Table 6.8: Subcells used for automation of the Column Encoder.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
Enc_nmos_col	$9.1\mu m \times 4.2\mu m$	Single nMOS transistor block whos drain is connected to an output line, gate connected to the an input column line, and source tied to ground.
Enc_nmos_dummy_col	$9.1\mu m \times 4.2\mu m$	Filler block such that various input and output signals remain connected appropriately.

6.6.6.1 Block Dimensions (Layout)

Given a column pitch of $pitch_{horiz}$. The *colEncoder* block dimensions are:

$Width = pitch_{horiz} \times num\_cols$

$Height = 4.2\mu m \times num\_output\_bits$

6.6.6.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL functions which automate layout, schematic, and symbol of this block:

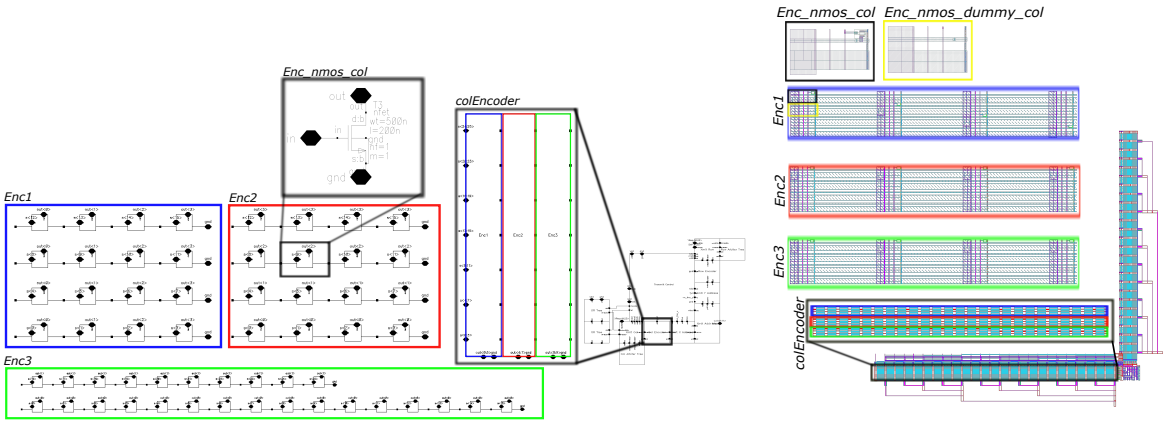
createXmitColEncoder_layout.m : Writes a SKILL function for generating the *colEncoder* layout.

createXmitColEncoder_schem.m : Writes a SKILL function for generating the *colEncoder* schematic.

createXmitColEncoder_symbol.m : Writes a SKILL function for generating the *colEncoder* symbol.

6.6.7 Transmit X Address [Final] Automation

The Transmit X Address [Final] block is responsible for latching the encoded column address to the Transmit Address Mux block such that it can be outputted from the transmitter. This latching is controlled by various signals from the Transmitter Control block. The Transmit X Address [Final] block is composed of arrangement of three different subcells. The first is *addrlatchX* which is directly coupled with the control signals *nco* and *txo* from the Transmitter Control block. Each *addrlatchX*



(a) Schematic (b) Layout

*Figure 6.17: Automated layout and schematic of the colEncoder block. The subencoders Enc*1*, Enc*2*, and Enc*3 *are only partially shown.*

block receives two input bits (from a pair of output bits from the colEncoder output). When signaled by the Transmitter Control, these input bits are latched to the output of the *addrlatchX* cell. However, it is important to note that depending on the number of columns in the processor array, not every output bit/line from the colEncoder may be dependent on the input column requests. Therefore, for these remaining bits, they are tied to Vdd. So, there exists $addrlatchX\_VDDB$ cell, which is used when both inputs should be tied to Vdd. There also exists an $addrlatchX\_VDDR$ cell, which is used when only the right (MSB) of the pair of bits should be tied to Vdd (See Fig. 6.18). The total number of *addrlatchX* blocks that should be placed side-by-side is:

$$num\_addrlatchx\_blocks = num\_output\_bits/2 \tag{6.10}$$

The second subcell used is $validneutral1in4$. This takes in two pairs of output bits from two $addrlatchX$ cells (total of 4 inputs). This subcell is responsible for checking that only one of the 4 inputs are high (valid 1-of-4 code). The total number of $validneutral1in4$ blocks placed side-by-side is:

$$num\_validneutral\_blocks = num\_addrlatchx\_blocks/2 \tag{6.11}$$

The output of each $validneutral1in4$ block is fed into a tree of C-element subcells. The C-element cells output logic high when both inputs are logic high and do not change state until both inputs go logic low (using state-holding elements). There are two different subcells for the C-elements, $Celem1$ and $Celem2$. The $Celem1$ block is used mostly however, on the first level of the tree, if it is an odd number of input then $Celem1$ is used. To conserve area, the C-element blocks are placed side-by-side even though they are connected in a tree-structure. There is automation for drawing metal wires which connect the C-elements in the tree-like structure (See Fig. 6.18). The output of the C-element tree is vnx which is connected to the Transmitter Control block. The total number of C-element blocks required is always:

$$num\_celem\_blocks = num\_validneutral\_blocks - 1 \tag{6.12}$$

The complete arrangement of these subcells in schematic and layout can be seen in Fig. 6.18.

Table 6.9: Subcells used for automation of the Transmit X Address [Final].

Subcells Used	**Dimensions ($W \times H$)**	**Description**
$addrlatchX$	$12.5\mu m \times 8.4\mu m$	Input is single pair of bits from colEncoder output bus. Latches this colEncoder output to output of the $XmitXAddr$ cell.
$addrlatchX\_VDDR$	$12.5\mu m \times 8.4\mu m$	Input is last bit (MSB) from colEncoder output bus. Second input is instead tied to Vdd. Latches this colEncoder output to output of the $XmitXAddr$ cell.
$addrlatchX\_VDDB$	$12.5\mu m \times 8.4\mu m$	Both inputs are tied to Vdd. Necessary when last two bits (MSB) are not dependent on row requests. Latches this colEncoder output to output of the $XmitXAddr$ cell.
$validneutral1in4$	$5.1\mu m \times 16.8\mu m$	Checks that 4 inputs represent valid 1-of-4 one-hot encoded group.
$Celem1$	$3.4\mu m \times 16.8\mu m$	C-element used when odd number of inputs (odd number of $validneutral1in4$ outputs).
$Celem2$	$3.4\mu m \times 16.8\mu m$	C-element used throughout most of C-element tree-structure.

6.6.7.1 Block Dimensions (Layout)

The $XmitXAddr$ block dimensions are:

$Width \approx 21\mu m$

$Height = 16.8\mu m \times num\_1of4\_groups$

6.6.7.2 MATLAB Files

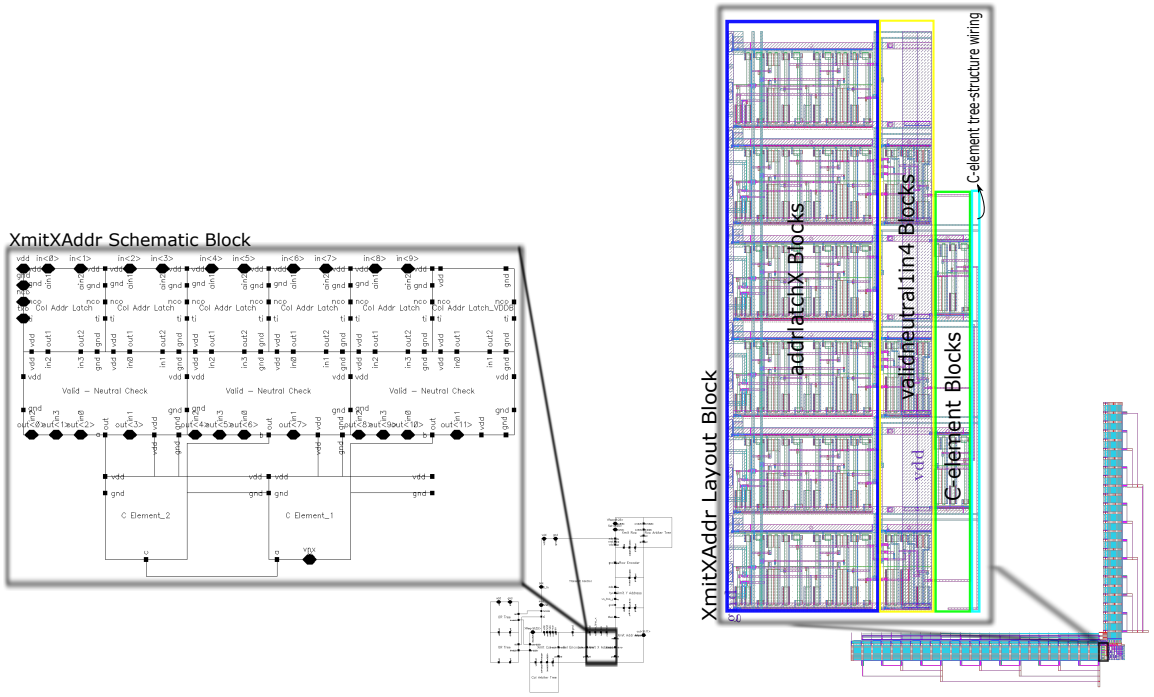
The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

$createXmitXAddr\_layout.m$: Writes a SKILL function for generating the $XmitXAddr$

layout.

createXmitXAddr_schem.m : Writes a SKILL function for generating the *XmitXAddr* schematic.

createXmitXAddr_symbol.m : Writes a SKILL function for generating the *XmitXAddr* symbol.



(a) Schematic (b) Layout

Figure 6.18: Automated layout and schematic of the XmitXAddr block.

6.6.8 OR Tree Automation

The OR Tree is responsible for checking if any inputs are logic high. There are two different OR Trees used both receiving inputs from $XmitCol$. The first OR Tree receives inputs from the $ncols < 0 : X >$ output from the $XmitCol$ block. This checks if any column request is high. The second OR Tree receives inputs from the $si < 0 : X >$ output from the $XmitCol$ block. This checks for the completion of a single column selection. The OR Tree automation is designed in a tree-like structure with its output at the top of the tree. It uses $ORgate$ subcells to automate the OR Tree. These $ORgate$ subcells are setup in a tree-like structure, however, they are stacked on top of each other such that Vdd, Gnd, and inputs from lower level of the tree can easily be connected to adjacent layers of the tree. To determine the number of OR gates in each layer, we simply start by the first layer is number of columns divided by 2, and each layer is divided by 2. If there is any odd number of OR gates in a layer, then the output of that OR gate is fed to the top layer. The number of layers can be computed by:

$$num\_layers = ceil(log_2(num\_columns)) \tag{6.13}$$

The placement of these layers and metal wire connections all occurs via MATLAB computation so that the complete OR Tree is automated (See Fig. 6.19).

Table 6.10: Subcells used for automation of the OR Tree.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
$ORgate$	$2.902\mu m \times 5.58\mu m$	Single OR gate used for automating the complete OR_Tree.

6.6.8.1 Block Dimensions (Layout)

Given a column pitch of $pitch_{horiz}$. The OR_tree block dimensions are:

$Width = pitch_{horiz} \times num\_cols$

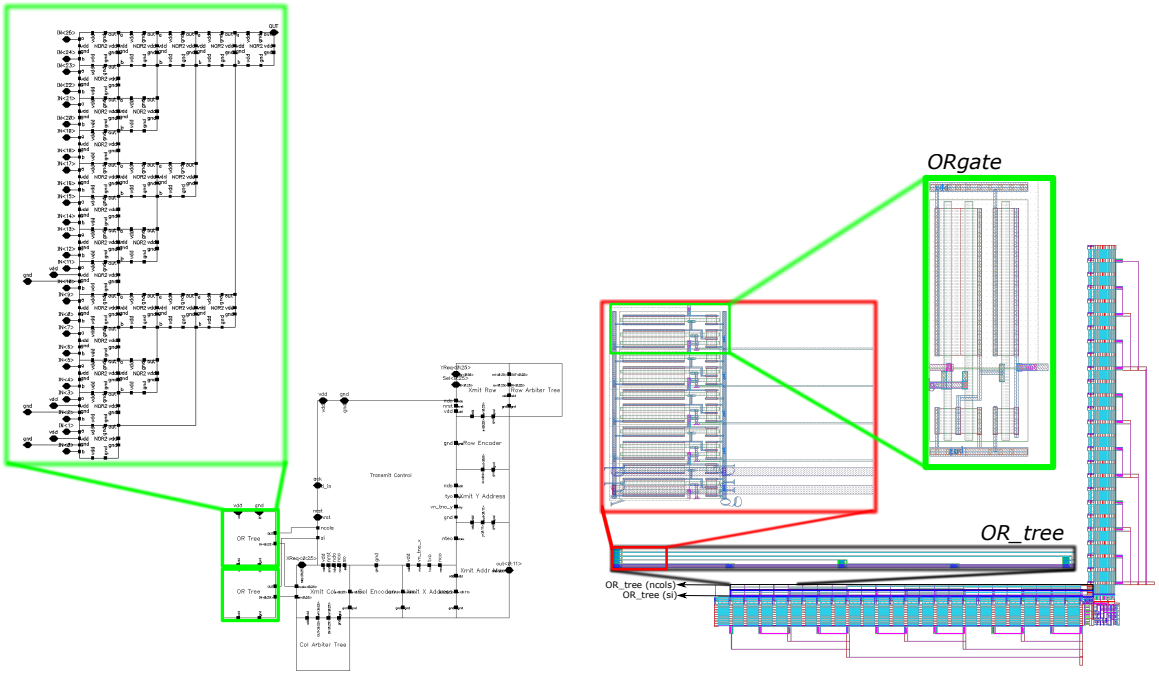
$Height = num\_layers \times 5.58\mu m$

6.6.8.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

$createOrTree\_layout\_schem.m$: Writes a SKILL function for generating the OR_tree schematic and layout. Calls $gen\_or\_tree.m$ function.

$createOrTree\_symbol.m$: Writes a SKILL function for generating the OR_tree symbol.



(a) Schematic (b) Layout

Figure 6.19: Automated layout and schematic of the OR_tree block.

6.6.9 OR Tree Stack Automation

The OR Tree Stack simply stacks the two OR Trees into one single layout cell. It also places bridge/filler cells for connection between the $XmitCol$ block and the $ORTree_Stack$. This block is also responsible for extending metal wires where necessary for connection between $XmitCol$ and the processor array column requests (See Fig. 6.20).

Table 6.11: Subcells used for automation of the OR Tree Stack.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
OR_tree	*See OR Tree Block*	Single OR_tree block.
$ortree\_xmitcol\_fill\_1$	$9.1\mu m \times 2.353\mu m$	Filler cell for connecting to one of pair of $colArbLogic$ cells to first layer of OR Trees.
$ortree\_xmitcol\_fill\_2$	$9.1\mu m \times 2.353\mu m$	Filler cell for connecting to second of pair of $colArbLogic$ cells to first layer of OR Trees.

6.6.9.1 Block Dimensions (Layout)

Given a column pitch of $pitch_{horiz}$. The $ORTree\_Stack$ block dimensions are:

$$Width = pitch_{horiz} \times num\_cols$$

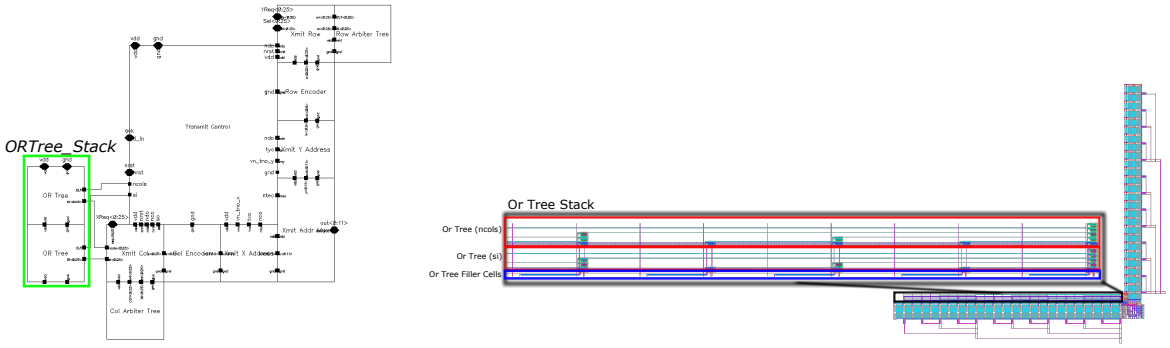
$$Height = num\_layers \times 5.58\mu m \times 2 + 2.353\mu m$$

6.6.9.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this

block:

createOrTreeStack_layout.m : Writes a SKILL function for generating the *ORTree_Stack* layout.



(a) Schematic (b) Layout

Figure 6.20: Automated layout and schematic of the ORTree_Stack block.

6.6.10 XmitYAddr and Row Encoder Bridge/Filler Automation

This block extends metal wires of the *Gnd* and output signals from the Row Encoder to the *XmitY Addr*. Considering it is only metal, it is only necessary for layout (no schematic or symbol). See Fig. 6.21 for this layout.

Table 6.12: Subcells used for automation of the xmityaddr_rowenc_bridge.

Subcells Used	Dimensions ($W \times H$)	Description
None	N/A	N/A

6.6.10.1 Block Dimensions (Layout)

The *xmityaddr_rowenc_bridge* block dimensions are:

$Width = num\_output\_bits \times 4.2\mu m$

$Height = 21.808\mu m$

6.6.10.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

$createXmitYAddr\_RowEncoder\_Bridge.m$: Writes a SKILL function for generating the *xmityaddr_rowenc_bridge* layout.

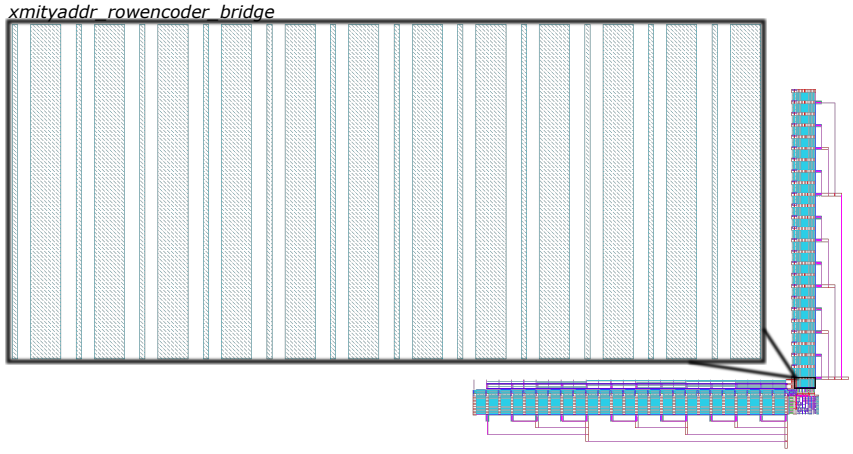


Figure 6.21: Bridge between XmitY Addr block and rowEncoder block.

6.6.11 XmitXAddr and Col Encoder Bridge/Filler Automation

This block extends metal wires of the *Gnd* and output signals from the Column Encoder to the *XmitXAddr*. Considering it is only metal, it is only necessary for layout (no schematic or symbol). See Fig. 6.22 for this layout.

Table 6.13: Subcells used for automation of the xmitxaddr_colenc_bridge.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
None	N/A	N/A

6.6.11.1 Block Dimensions (Layout)

The *xmitxaddr_colenc_bridge* block dimensions are:

$Width = 3.564\mu m$

$Height = num\_output\_bits \times 4.2\mu m$

6.6.11.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

createXmitXAddr_ColEncoder_Bridge.m : Writes a SKILL function for generating the *xmitxaddr_colenc_bridge* layout.

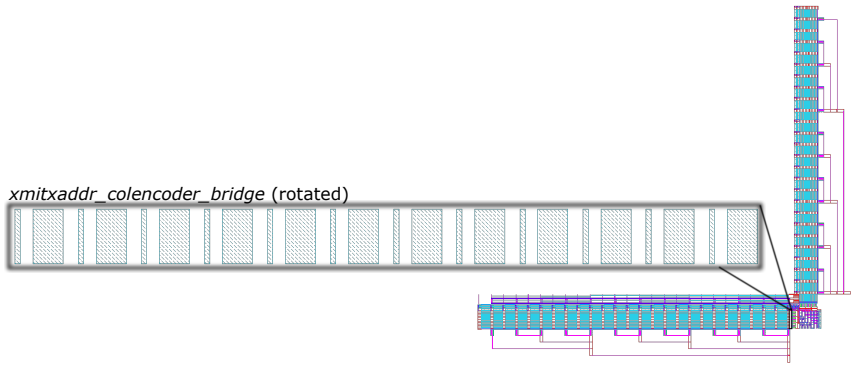


Figure 6.22: Bridge between XmitXAddr block and colEncoder block.

6.6.12 Transmitter Control Top Bridge/Filler Automation

This block extends metal wire connections from the Transmitter Control to the two OR Trees as well as the $XmitRow$ block. It acts as a bridge between these blocks. However, this changes depending on the height or the $ORTree\_Stack$. The automation is important for automatically determining the height of the $ORTree\_Stack$ and then generating this filler/bridge accordingly. This can be seen in Fig. 6.23.

Table 6.14: Subcells used for automation of the xmitcontrol_top_fill.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
$xmitcontrol\_top\_fill\_1a$	$26.1\mu m \times 2.416\mu m$	Filler cell placed at the bottom of the complete $xmitcontrol\_top\_fill$. It connects to the first (bottom OR Tree for si).
$xmitcontrol\_top\_fill\_1b$	$26.1\mu m \times 2.416\mu m$	Filler cell placed at the bottom of the complete $xmitcontrol\_top\_fill$. It is filler metal to bridge gap of bottom OR Tree (si) height.
$xmitcontrol\_top\_fill\_2a$	$26.1\mu m \times 3.902\mu m$	Filler cell placed at adjacent to the bottom of the second OR Tree ($ncols$).
$xmitcontrol\_top\_fill\_2b$	$26.1\mu m \times 2.416\mu m$	Filler cell placed at the adjacent to height of second OR Tree ($ncols$).
$xmitcontrol\_top\_fill\_3$	$26.1\mu m \times 0.602\mu m$	Filler cell for very top of this $xmitcontrol\_top\_fill$.

6.6.12.1 Block Dimensions (Layout)

Given num_layers as the number of layers in the OR Tree, the $xmitcontrol\_top\_fill$ block dimensions are:

$Width = 26.1\mu m$

$Height = 2.416\mu m \times (num\_layers - 1) \times 2 + 2.416\mu m + 3.902\mu m$

6.6.12.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

$createXmitControl\_Top\_Bridge.m$: Writes a SKILL function for generating the $xmitcontrol\_top\_fill$ layout.

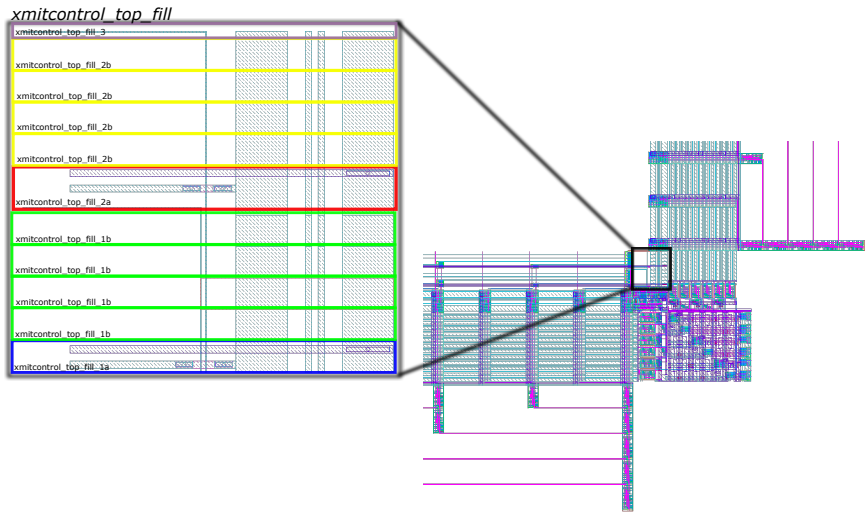


Figure 6.23: Bridge between $XmitControl\_Final$ block and $XmitRow$ and $ORTree_8stack$ block.

6.6.13 Transmit Address Mux Automation

The Transmit Address Mux block receives two input buses, one from the $XmitXAddr$ output and one from the $XmitYAddr$ output. It outputs the row or column 1-of-4 encoded address when appropriate. It also outputs the tailword when signaled by the Transmitter Control block. The automation consisted of placing the left side of this block using subcells $addrmux_left1$ and $addrmux_left2$. This is to connect with the $XmitXAddr$ block. The next step involves placing the top cells, $addrmux_top1$ and $addrmux_top2$, for connecting with the $XmitYAddr$ block. Finally, the $addrMux$ subcell is placed in a diagonal manner followed by row filler blocks ($addrMux_rowFill$) and column filler blocks ($addrMux_colFill$). When there is more rows than columns or more columns than rows, the inputs must be connected to ground. Therefore, the $addrMux\_rowFill\_gnd$ or $addrMux_colFill_gnd$ would be used. Finally, on the right side of this block the $out1in4tail$ block is placed which either outputs the row or column address, or outputs the tailword when signaled by the $XmitControl_Final$. This can be seen in Fig. 6.24.

6.6.13.1 Block Dimensions (Layout)

Given $num\_1of4groups\_rows$ and $num\_1of4groups\_cols$ number of 1-of-4 groups for rows and columns. The $XmitMuxAddr$ block dimensions are:

Table 6.15: Subcells used for automation of the Transmit Address Mux.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
$addrMux$	$16.8\mu m \times 16.8\mu m$	Block connected to single 1-of-4 group output from both $XmitYAddr$ and $XmitXAddr$. Placed diagonally in the $XmitAddrMux$ block.
$addrMux_rowFill$	$16.8\mu m \times 16.8\mu m$	Serves as bridge for remaining of row of 1-of-4 groups so that there is connection to $out1in4tail$.
$addrMux_colFill$	$16.8\mu m \times 16.8\mu m$	Serves as bridge for remaining of columns of 1-of-4 groups so that there is connection from $XmitXAdd$ output to its appropriate $addrMux$ cell.
$addrMux\_rowFill\_gnd$	$16.8\mu m \times 16.8\mu m$	Serves as bridge for remaining of row of 1-of-4 groups so that there is connection to $out1in4tail$. Used when more column 1-of-4 groups than rows so the outputs are grounded.
$addrMux_colFill$	$16.8\mu m \times 16.8\mu m$	Serves as bridge for remaining of columns of 1-of-4 groups so that there is connection from $XmitXAdd$ output to its appropriate $addrMux$ cell. Used when more row 1-of-4 groups than columns so the outputs are grounded.
$addrMux_left1$	$4.936\mu m \times 16.8\mu m$	Filler for left side of $XmitAddrMux$ block to bridge gap between $XmitXAddr$ and $XmitAddrMux$. Placed at beginning of every row except for last.
$addrMux_left2$	$4.936\mu m \times 16.8\mu m$	Filler for left side of $XmitAddrMux$ block to bridge gap between $XmitXAddr$ and $XmitAddrMux$. Placed at last/bottom row only.
$addrMux_top1$	$16.8\mu m \times 6.841\mu m$	Filler for top of $XmitAddrMux$ block to bridge gap between $XmitYAddr$ and $XmitAddrMux$. Placed at beginning/top of every column except for last.
$addrMux_top2$	$10.828\mu m \times 6.841\mu m$	Filler for top of $XmitAddrMux$ block to bridge gap between $XmitYAddr$ and $XmitAddrMux$. Placed at beginning/top of last/right column only.
$out1in4tail$	$10.828\mu m \times 16.8\mu m$	Receives output from $addrMux$ block and either outputs that row/column address or the tailword depending on its input $nteo$ from the Transmitter Control block. Placed as last block in each row of $XmitAddrMux$.

$Width = 4.936\mu m + max(num\_1of4groups\_rows, num\_1of4groups\_cols) \times 16.8\mu m$

$Height = 6.841\mu m + max(num\_1of4groups\_rows, num\_1of4groups\_cols) \times 16.8\mu m$

6.6.13.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

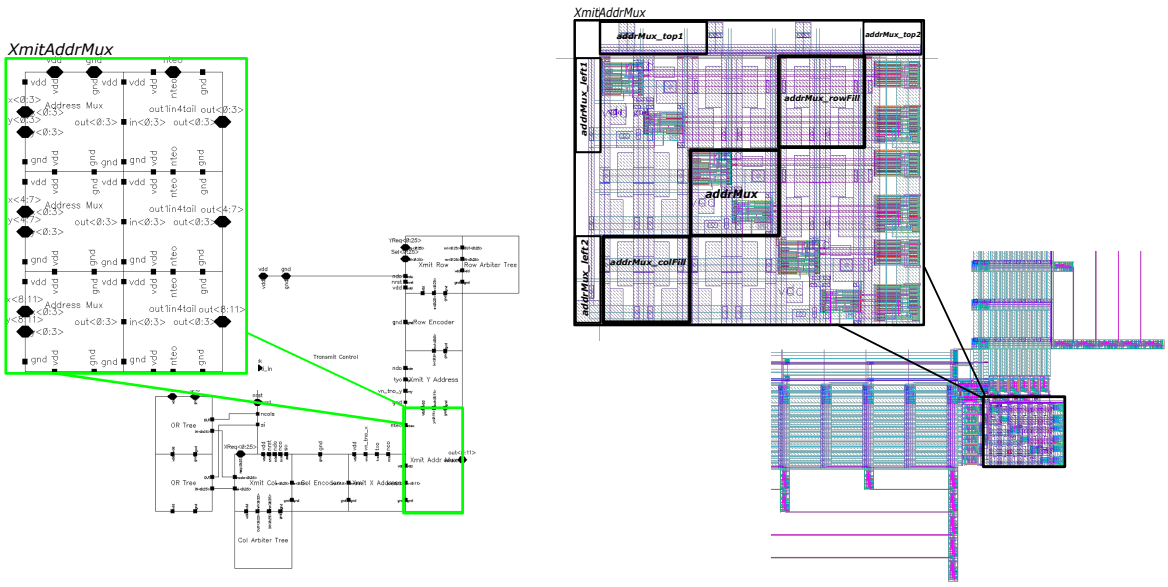
$createXmitAddrMux\_layout.m$: Writes a SKILL function for generating the $XmitAddrMux$ layout.

$createXmitAddrMux\_schem.m$: Writes a SKILL function for generating the $XmitAddrMux$ schematic.

$createXmitAddrMux\_symbol.m$: Writes a SKILL function for generating the $XmitAddrMux$ symbol.

6.6.14 Row Arbiter Tree Automation

The Row Arbiter Tree is responsible for selecting a single row request when multiple row requests are present (arbitration). It is similar to the OR tree in that it is constructed in a tree-like structure using the *Arbcell* subcell. There is also filler cells (*arbtreerow_bridge*) to the left of the Row Arbiter such that there is a bridge



(a) Schematic (b) Layout

Figure 6.24: Automated layout and schematic of the XmitAddrMux block.

between the Row Arbiter and the Row Encoder. The actual row arbiter tree cell without these filler cells is automated independently and has dimensions:

$$Width = num\_layers \times 18.64\mu m$$

$$Height = pitch_{vert} \times 9.1\mu m$$

See Fig. 6.25 for visual representation of the Row Arbiter.

Table 6.16: Subcells used for automation of the RowArbiter.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
Arb_tree_row	See Above	Complete generated arbiter tree without filler cells.
Arbcell	$18.64\mu m \times 7.58\mu m$	Single arbiter cell for selecting one of two inputs.
arbtreerow_bridge1	$2.928\mu m \times 9.1\mu m$	Bridge between Row Arbiter and Row Encoder.
arbtreerow_bridge2	$2.928\mu m \times 9.1\mu m$	Bridge between Row Arbiter and Row Encoder. Placed at end/top of tree.

6.6.14.1 Block Dimensions (Layout)

Given a row pitch of $pitch_{vert}$. The *RowArbiter* block dimensions are:

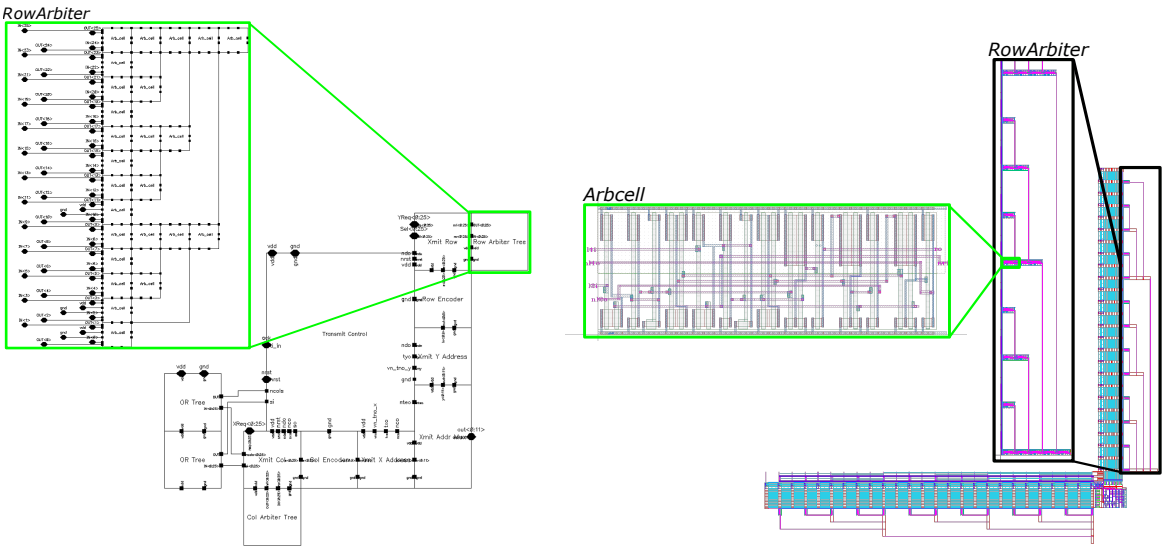
$$Width = 2.928\mu m + num\_layers \times 18.64\mu m$$

$$Height = pitch_{vert} \times 9.1\mu m$$

6.6.14.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

createArbiterTreeRow_layout_schem.m : Writes a SKILL function for generating the Arb_tree_row layout and schematic.
createArbiterTreeRow_symbol : Writes a SKILL function for generating the *RowArbiter* symbol. *createXmitRowArbiter_layout* : Writes a SKILL function for generating complete *RowArbiter* layout.



(a) Schematic (b) Layout

Figure 6.25: Automated layout and schematic of the RowArbiter block.

6.6.15 Column Arbiter Tree Automation

The Column Arbiter Tree is responsible for selecting a single column request when multiple column requests are present (arbitration). It is similar to the OR tree in that it is constructed in a tree-like structure using the *Arbcell* subcell. There is also filler cells (*arbtreecol_bridge*) at the bottom of the Column Arbiter such that there is a bridge between the Column Arbiter and the Column Encoder. The actual column arbiter tree cell without these filler cells is automated independently and has dimensions:

$Width = pitch_{horiz} \times 9.1\mu m$

$Height = num\_layers \times 18.64\mu m$

See Fig. 6.26 for visual representation of the Column Arbiter.

Table 6.17: Subcells used for automation of the ColArbiter.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
Arb_tree_column	See Above	Complete generated arbiter tree without filler cells.
Arbcell	$18.64\mu m \times 7.58\mu m$	Single arbiter cell for selecting one of two inputs.
*arbtreecol_bridge*1	$2.928\mu m \times 9.1\mu m$	Bridge between Column Arbiter and Column Encoder.
*arbtreecol_bridge*2	$2.928\mu m \times 9.1\mu m$	Bridge between Column Arbiter and Column Encoder. Placed at end/left of tree.

6.6.15.1 Block Dimensions (Layout)

Given a row pitch of $pitch_{horiz}$. The *ColArbiter* block dimensions are:

$Width = pitch_{horiz} \times 9.1\mu m$

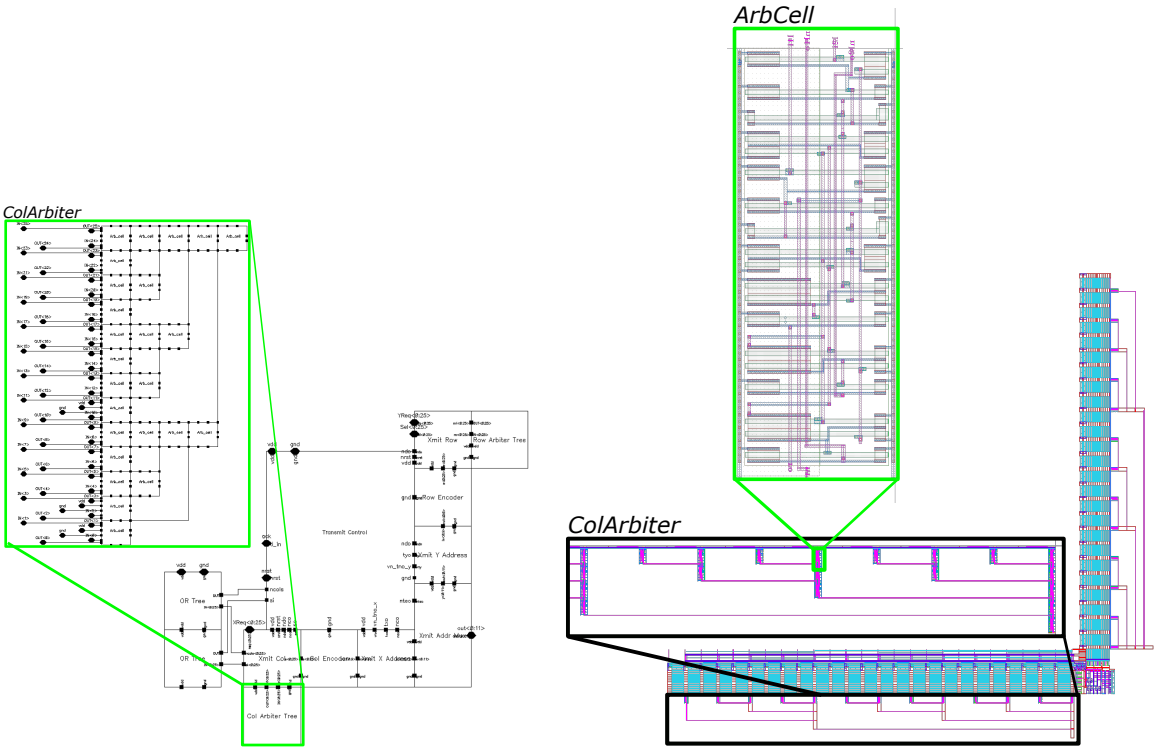
$Height = 2.928\mu m + num\_layers \times 18.64\mu m$

6.6.15.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

createArbiterTreeCol_layout_schem.m : Writes a SKILL function for generating the *Arb_tree_col* layout and schematic.

createArbiterTreeCol_symbol : Writes a SKILL function for generating the *ColArbiter* symbol. *createXmitColArbiter_layout* : Writes a SKILL function for generating complete *ColArbiter* layout.



(a) Schematic (b) Layout

Figure 6.26: Automated layout and schematic of the ColArbiter block.

6.6.16 Transmitter Control Automation

The transmitter control block is "heart" of the transmitter. It is responsible for controlling row and column requests, encoding the row/column address-event requests into 1-of-4 codes, and then latching these encoded address-events to the

output bus in their appropriate order. The transmitter control block is composed of asynchronous blocks for controlling the timing of these operations. It ensures as address-event requests are received, the encoded row address is outputted, followed by the appropriate column addresses, followed by the tailword. See Fig. 6.27 for the schematic and layout of the transmitter control. This transmitter control block is not dependent on the number of rows nor the number of columns of the processor array. Henceforth, it is simply a block in the automation library that must be placed in the correct position of the transmitter so that it properly connects to other blocks of the transmitter. This block name is $XmitControl$. It is composed of various asynchronous blocks also included in this AER Automation library. However, there are various metal wires that must be drawn via SKILL scripts at a length that is dependent on the number of rows and columns (see list of wires below). This is because the blocks that the transmitter are connected to do vary in size depending on the number of rows and columns. Therefore, these metal wire lengths are dependent on the size of connecting blocks such that the wires connect appropriately. The final generated transmit control block with these automatically drawn metal wires is called $XmitControl\_Final$.

Table 6.18: Subcells used for automation of the Transmitter Control.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
$XmitControl$	$26.1\mu m \times 18.8\mu m$	Main Transmitter Control block without any automated metal extensions.

6.6.16.1 Block Dimensions (Layout)

The $XmitControl\_Final$ block dimensions are:

$Width = 26.1\mu m$

$Height \approx 18.8\mu m$

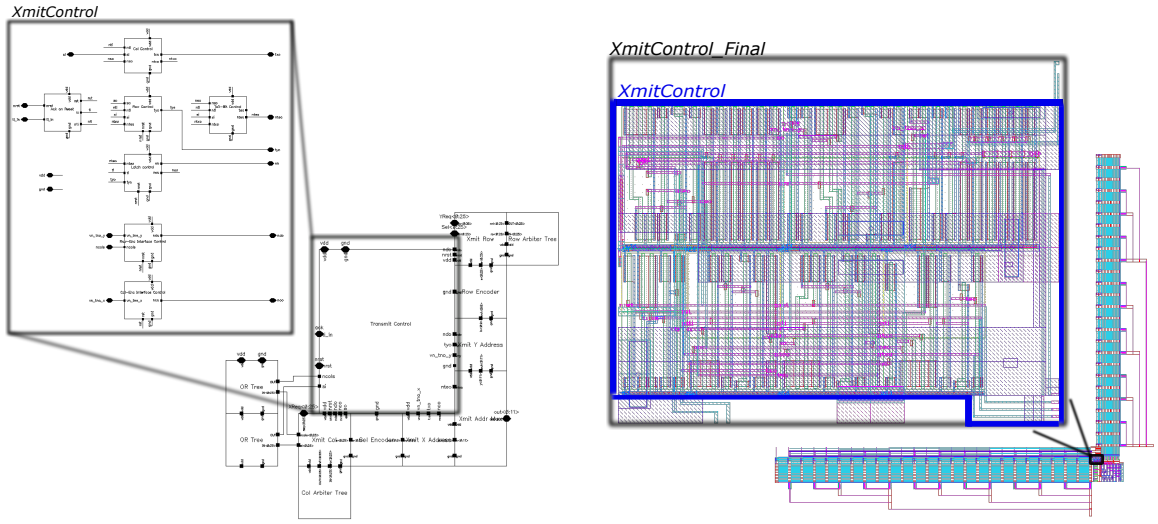
6.6.16.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

$createXmitControl\_layout.m$: Writes a SKILL function for generating the $XmitControl\_Final$ layout.

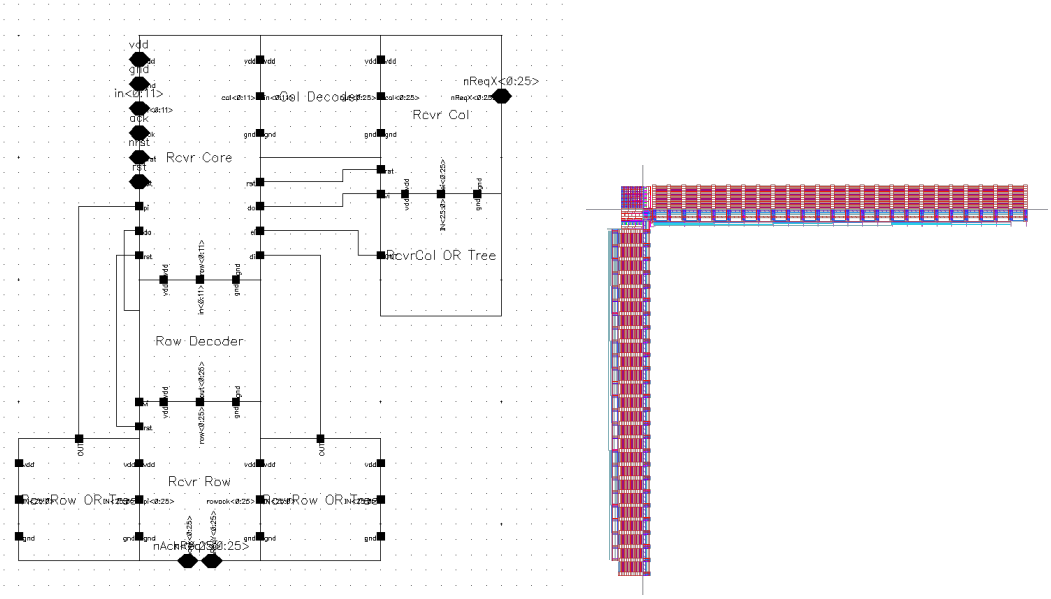
6.7 Receiver Automation

The complete receiver (layout, schematic, and symbol) is automatically generated given the number of rows (even only), number of columns (even only), and the row and column pitch of the user's event-based processing element array. The receiver's automation is a piece-wise process. It involves first the automation of the "major blocks"



(a) Schematic (b) Layout

Figure 6.27: Automated layout and schematic of the XmitControl_Final block.



(a) Schematic (b) Layout

Figure 6.28: Schematic and layout of the complete delay-insensitive receiver.

involved in the receiver design. Finally, it involves the proper arrangement/connection of these blocks to create the complete functioning receiver. All of the subcells used for generating these major blocks are located in the *AER Automation* library. As previously noted, MATLAB is used for holding all of the necessary parameters, performing various computations for generating accurate SKILL functions that create all of the major blocks and arrangement of these blocks to form the complete receiver in schematic and layout. In the proceeding sections, we will discuss the how each of the major blocks involved in the automation process are generated. The complete receiver schematic and layout can be seen in Fig. 6.28. It is composed of the major blocks (including filler cells) that were automated and then interfaced for generating the complete transmitter. These blocks are listed in Table 6.19. In regards to the receiver section of this report, first, the signals and timing of the signals required for interfacing with the receiver will be discussed. Then, constraints/specifications and simulations results will be discussed. Finally, the automation of each of the major blocks above along with the generation of the complete receiver will be discussed.

6.7.1 Interfacing Receiver with an Address-Event-Based Processor Array

The receiver automation is independent of the user's address-event-based processor array. However, the communication protocol of the receiver must be considered

Table 6.19: List of major blocks that were automated and used for the complete receiver generation.

Receiver Block	**Cell Name (in Cadence)**	**Description**
Receiver Row OR Tree Stack	$ORTree\_Stack\_RcvrRow$	OR Tree used for feedback to Receiver Control for signaling when the row address has been latched. Second OR Tree used for signaling to Receiver Control that acknowledge has been received. Also includes filler/bridge cells.
Receiver Column OR Tree Stack	$ORTree\_Stack\_RcvrCol$	OR Tree used for feedback to Receiver Control for signaling when the column address has been latched. Also includes filler/bridge cells.
Receiver X Address	$RcvrXAddr$	Receives incoming column address and latches column address to the column output of the Receiver Core when signaled by the Receiver Control.
Receiver Y Address A	$RcvrYAddrA$	Receives incoming row address and latches row address to $RcvrYAddrB$ when signaled by the Receiver Control.
Receiver Y Address B	$RcvrYAddrA$	Receives incoming row address and latches row address to the row output of the Receiver Core when signaled by the Receiver Control.
Receiver Control	$rcvrcontrol\_final$	Final Receiver Control block after metal extensions have been automated.
Receiver Column	$RcvrCol$	Receives decoded column address to send column request to appropriate column.
Receiver Row	$RcvrRow$	Receives decoded row address to send row request to appropriate row.
Column Decoder	$colDecoder$	Decodes received column address from 1-of-4 one-hot encoding to its selected column request.
Row Decoder	$rowDecoder$	Decodes received row address from 1-of-4 one-hot encoding to its selected row request.
Receiver Core	$RcvrCore$	Handles controlling the incoming row and column addresses and latching these addresses appropriately.
Receiver Input Corner	$RcvrInCorner$	Input bus connection to both the $RcvrXAddr$ and $RcvrYAddrA$.

when designing the interface between the user's processor array and the transmitter. Each processor element in the array must contain specific signals and the control of these signals must operate in a specific way for proper communication with the receiver. The block diagram of the receiver and processor array interface can be seen in Fig. 6.29. Assuming Y number of rows and X number of columns, the signals for interfacing the receiver with the processor array can be seen in 6.20.

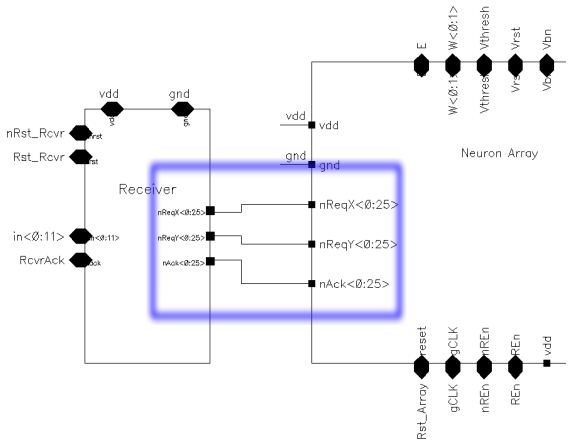


Figure 6.29: Receiver interface with address-event-based processor array. The blue box surrounds the three signals used for communication with the receiver.

Table 6.20: Receiver Signals for Interfacing with Processor Array (X Columns, Y Rows)

Receiver Signal	Direction	Description
$nReqX < 0 : X >$	Output	Column request to processor array (active low signal)
$nReqY < 0 : Y >$	Output	Row request to processor array (active low signal)
$nAck < 0 : Y >$	Input	Acknowledge from processor array (active low signal)

Therefore, each event-based processing element in the array must have a row request input ($nReqY$), column request input ($nReqX$), and acknowledge output

($nAck$). Each row of processing elements of the array should be tied to a single $nReqY$ wire (wired-OR). Each column of processing elements of the array should be tied to a single $nReqX$ wire (wired-OR). Considering the receiver architecture is asynchronous, the user has flexibility in the control of these signals. These signals can be either synchronous or asynchronous. However, the order of assertion of these signals is important for proper communication between the receiver and the processor array. The order of operations is as follows:

1. Receiver output $nReqY$ is asserted low.
2. Receiver output $nReqX$ is asserted low.
3. Processor element $nAck$ signal is asserted low.
4. Signals $nReqY$ and $nReqX$ is deasserted (logic high).
5. Signal $nAck$ should be deasserted (logic high).

6.7.1.1 Processing Element to Receiver Communication Block

To further facilitate the communication, a block was designed that can be placed within each processor element that acts as a bridge between the event-based processor element and the receiver. This block is called $rcvrinterface$ and is placed within each element. The schematic of this block can be seen in Fig. 6.30. It is a NOR gate such that when both $nReqY$ and $nReqX$ are active low (controlled by the receiver), the output is pulled high. This is a signal to your processor element (or neuron) that the cell has received an event. This can signal other blocks in the processor element but

it should also turn on the nMOS transistor which pulls *nAck* signal low. This is the acknowledge signal back to the receiver.

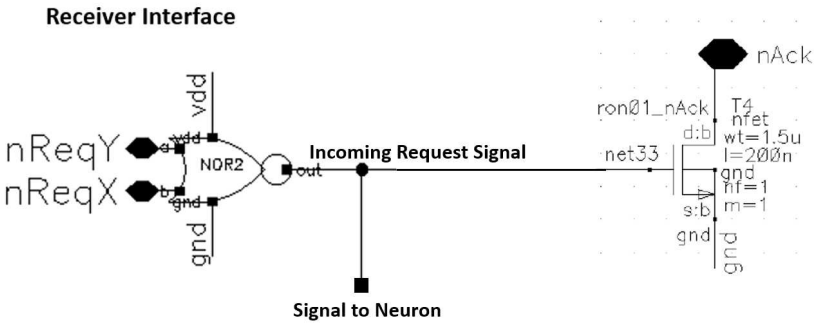


Figure 6.30: Schematic of the block placed within each processor element cell to facilitate communication between the event-based processor and the receiver.

6.7.2 Receiver Row OR Tree Stack Automation

The Receiver OR Tree Stack is comprised of two identical OR trees stacked and placed along the vertical/left side of the receiver. One OR Tree is responsible for checking if a final row request has been signaled to the array. The second OR Tree is responsible for checking if an acknowledge signal has been received by the array. The first steps in automation involve generating the OR Trees. This is done identically to that in the transmitter. It is designed in a tree-like structure composed of *ORgate* subcells. The number of layers in the OR tree is:

$$num\_layers = ceil(log_2(num\_columns)) \tag{6.14}$$

Given a row pitch of $pitch_{vert}$. The OR_tree block dimensions are:

$$Width = num\_layers \times 5.58\mu m$$

$$Height = pitch_{vert} \times num\_rows$$

These two OR Trees are stacked on top of each other and placed vertically. There is also a filler layer composed of filler cells ($ortree\_rcvrrow\_fill1$ and $ortree\_rcvrrow\_fill1$) serving as a bridge between the $RcvrRow$ and the $ORTree\_Stack\_RcvrRow$. This can be seen in Fig. 6.31.

Table 6.21: Subcells used for automation of the Row OR Tree Stack.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
$OR\_tree\_rcvrrow$	See Above	Complete single OR Tree automated independently.
$ORgate$	$5.58\mu m \times 2.902\mu m$	Single OR gate used for automating the complete $OR\_tree\_rcvrrow$.
$ortree_rcvrrow_fill1$	$2.381\mu m \times 9.1\mu m$	Filler cell for bridge between OR Tree and $RcvrRow$ for all rows in line with $ORgate$ cell.
$ortree\_rcvrrow\_fill2$	$2.381\mu m \times 9.1\mu m$	Filler cell for bridge between OR Tree and $RcvrRow$ for all other rows not inline with $ORgate$ cell.

6.7.2.1 Block Dimensions (Layout)

Given a row pitch of $pitch_{vert}$. The $ORTree\_Stack\_RcvrRow$ block dimensions are:

$$Width = num\_layers \times 5.58\mu m \times 2 + 2.381\mu m$$

$Height = pitch_{vert} \times num\_rows$

6.7.2.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

createRcvrRowOrTree_layout_schem.m : Writes a SKILL function for generating the *OR_tree_rcvrrow* layout and schematic.

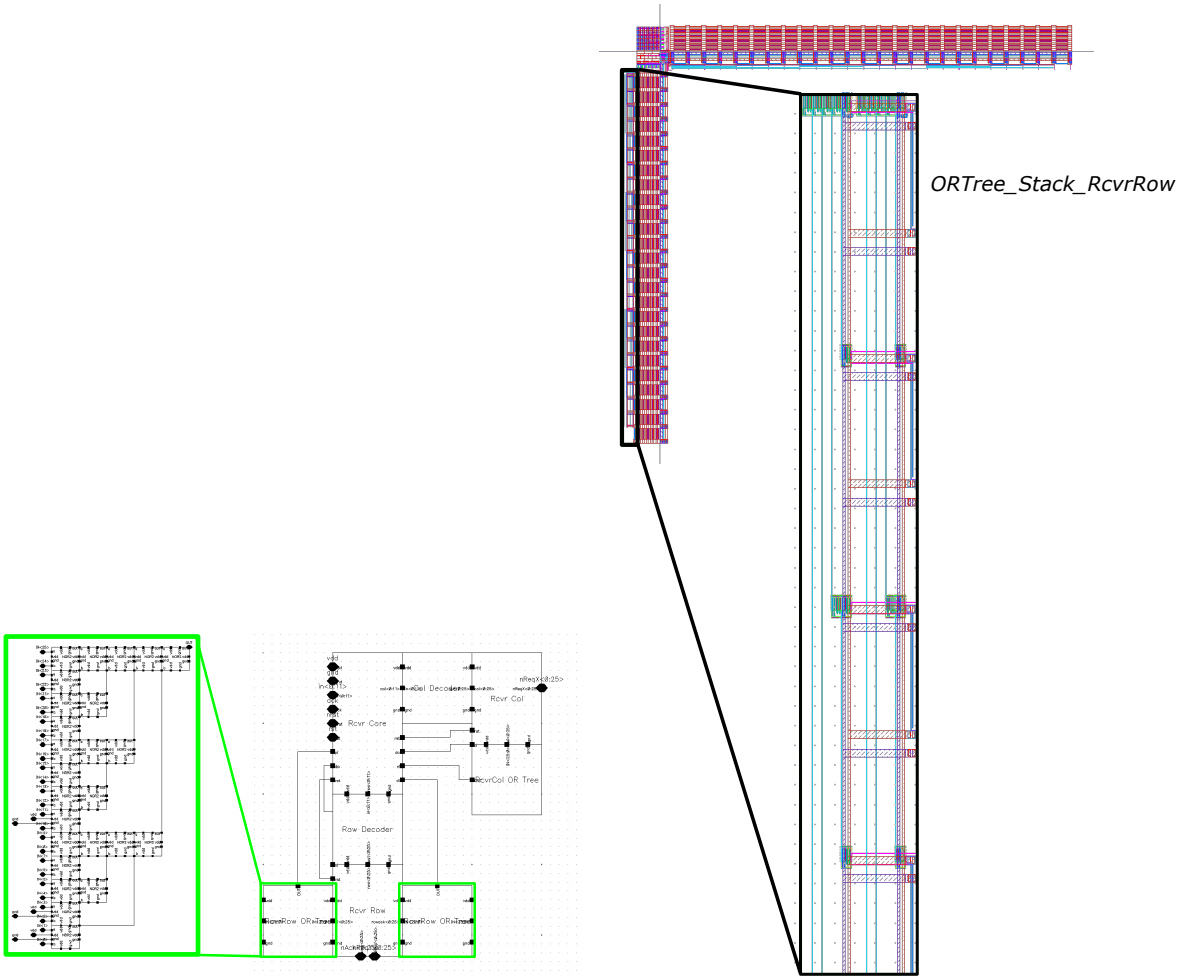
createRcvrRowOrTree_symbol.m : Writes a SKILL function for generating the *ORTree_Stack_*

symbol.

createRcvrRowOrTreeStack_layout.m : Writes a SKILL function for generating the *ORTree_Stack_RcvrRow* layout.

6.7.3 Receiver Column OR Tree Stack Automation

The Receiver OR Tree Stack is comprised of a single OR Tree for checking if the column request coupled to the array has been selected/asserted. The first step in automation involve generating the OR Tree. This is done identically to that in the



(a) Schematic (b) Layout

Figure 6.31: Automated layout and schematic of the ORTree_Stack_RcvrRow block.

transmitter. It is designed in a tree-like structure composed of $ORgate$ subcells. The number of layers in the OR tree is:

$$num\_layers = ceil(log_2(num\_columns)) \tag{6.15}$$

Given a column pitch of $pitch_{horiz}$. The OR_tree block dimensions are:

$Width = pitch_{horiz} \times num\_cols$

$Height = num\_layers \times 5.58\mu m$

There is also a filler layer composed of filler cells ($ortree\_rcvrcol\_fill$) serving as a bridge between the $RcvrCol$ and the $ORTree\_Stack\_RcvrCol$. This can be seen in Fig. 6.32.

Table 6.22: Subcells used for automation of the Column OR Tree Stack.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
$OR\_tree\_rcvrcol$	See Above	Complete single OR Tree automated independently.
$ORgate$	$2.902\mu m \times 5.58\mu m$	Single OR gate used for automating the complete $OR\_tree\_rcvrcol$.
$ortree\_rcvrcol\_fill1$	$9.1\mu m \times 2.381\mu m$	Filler cell for bridge between OR Tree and $RcvrCol$ for all columns in line with $ORgate$ cell.
$ortree\_rcvrcol\_fill2$	$9.1\mu m \times 2.381\mu m$	Filler cell for bridge between OR Tree and $RcvrCol$ for all other columns not inline with $ORgate$ cell.

6.7.3.1 Block Dimensions (Layout)

Given a column pitch of $pitch_{horiz}$. The $ORTree\_Stack\_RcvrCol$ block dimensions are:

$Width = pitch_{horiz} \times num\_cols$ $Height = num\_layers \times 5.58\mu m \times 2 + 2.381\mu m$

6.7.3.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

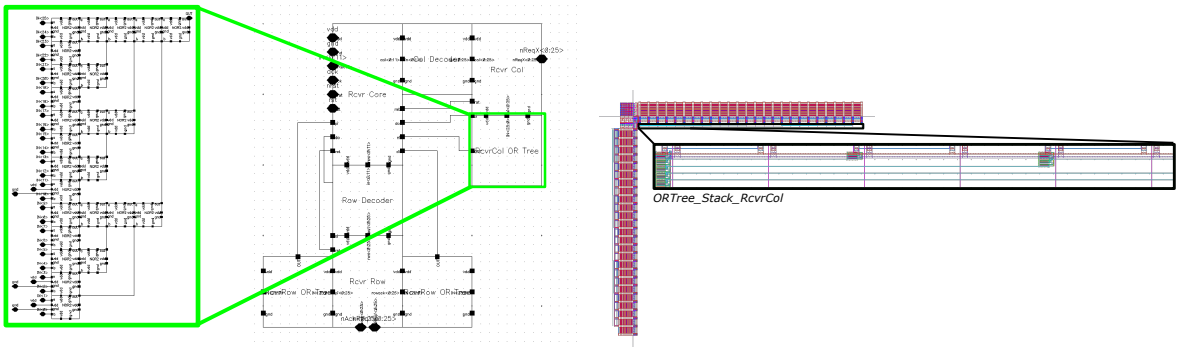
createRcvrColOrTree_layout_schem.m : Writes a SKILL function for generating the *OR_tree_rcvrcol* layout and schematic.

createRcvrColOrTree_symbol.m : Writes a SKILL function for generating the *ORTree_Stack_R* symbol.

createRcvrColOrTreeStack_layout.m : Writes a SKILL function for generating the *ORTree_Stack_RcvrCol* layout.

6.7.4 Receiver X Address Automation

The Receiver X Address block is responsible for latching the incoming 1-of-4 encoded column address when signaled by the Receiver Control block. It is one of the blocks that make up the complete Receiver Core. It consists of first *rcvraddrlatchX*



(a) Schematic (b) Layout

Figure 6.32: Automated layout and schematic of the ORTree_Stack_RcvrCol block.

subcells which are directly connected to the incoming address. It is responsible for latching the address when signaled by the Receiver Control. The number of $rcvraddrlatchX$ cells is equal to $2 \times num\_1of4groups$. The $num_1of4groups$ value is always the $max(num_1of4groups_row, num_1of4groups_col)$. The output of the $rcvraddrlatchX$ cells is tied to the input of the $validneutral1in4$ subcell. There is one $validneutral1in4$ subcell for every 1-of-4 group. Finally, the C-element tree is placed at the output of the $validneutral1in4$ cells using subcells $Celem\_single$, $Celem\_last$, $Celem0$, $Celem1$ and $Celem2$ (similar to that in the transmitter). There is also exists a filler cells for bridging game between the Receiver Column Decoder and the Receiver X Address block ($RcvrXAddr\_bridge$). See Fig. 6.33 for further visualization of this.

Table 6.23: Subcells used for automation of the Receiver X Address.

Subcells Used	Dimensions ($W \times H$)	Description
$rcvraddrlatchX$	$8.4\mu m \times 9.0\mu m$	Cell for latching incoming address.
$validneutral1in4$	$16.8\mu m \times 5.1\mu m$	Valid-Neutral cell for checking that incoming address is valid or neutral state.
$Celem\_last$	$16.8\mu m \times 3.4\mu m$	Used in building C-element tree.
$Celem\_single$	$16.8\mu m \times 3.4\mu m$	Used in building C-element tree.
$Celem0$	$16.8\mu m \times 3.4\mu m$	Used in building C-element tree.
$Celem1$	$16.8\mu m \times 3.4\mu m$	Used in building C-element tree.
$Celem2$	$16.8\mu m \times 3.4\mu m$	Used in building C-element tree.
$RcvrXAddr\_bridge$	$16.8\mu m \times 1.0\mu m$	Bridge between $RcvrXAddr$ and $ColumnDecoder$.

6.7.4.1 Block Dimensions (Layout)

The $RcvrXAddr$ block dimensions are:

$Width \approx 17.5\mu m$

$Height = num\_1of4groups \times 16.8\mu m$

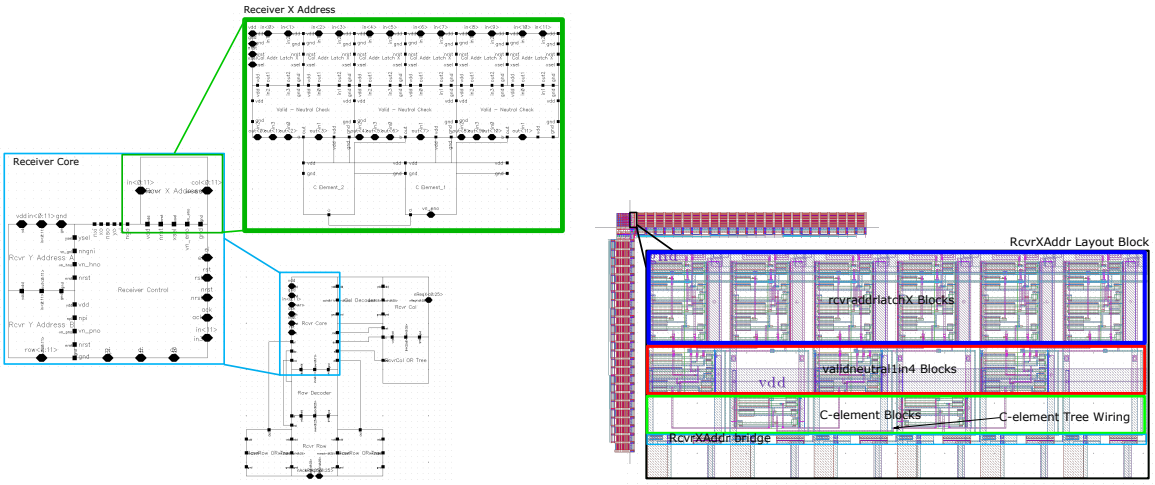
6.7.4.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

$createRcvrXAddr\_layout.m$: Writes a SKILL function for generating the $RcvrXAddr$ layout.

createRcvrXAddr_schem.m : Writes a SKILL function for generating the *RcvrXAddr* schematic.

createRcvrXAddr_symbol.m : Writes a SKILL function for generating the *RcvrXAddr* symbol.



(a) Schematic (b) Layout

Figure 6.33: Automated layout and schematic of the RcvrXAddr block.

6.7.5 Receiver Y Address A Automation

The Receiver Y Address A block is responsible for latching the incoming 1-of-4 encoded row address when signaled by the Receiver Control block. It is one of the blocks that make up the complete Receiver Core. It consists of first *rcvraddrlatchYA* subcells which are directly connected to the incoming address. It is responsible for latching the address when signaled by the Receiver Control. The number of

$rcvraddrlatchYA$ cells is equal to $2\times num\_1of4groups$. The output of the $rcvraddrlatchYA$ cells is tied to the input of the $validneutral1in4$ subcell. There is one $validneutral1in4$ subcell for every 1-of-4 group. Finally, the C-element tree wiring is placed at the output of the $validneutral1in4$ cells using subcells $Celem\_single$, $Celem\_last$, $Celem0$, $Celem1$ and $Celem2$ (similar to that in the transmitter). Unlike the other Receiver Address blocks, the $RcvrYAddrA$ block consists also of a second independent "CTree" ($RcvrYAddrA\_ctree$) consisting of only $validneutral1in4\_rcvrYA$ and C-element blocks. The input to these blocks comes directly from the input 1-of-4 encoded address, and then also is tied to the $rcvraddrlatchYA$ blocks as well. The dimensions of this second CTree are:

$Width = num\_1ofgroups \times 16.8\mu m$

$Height \approx 8.5\mu m$

There is also exists a filler cells for bridging game between the $RcvrYAddrA$ and the $RcvrYAddrB$ block ($RcvrYAddrA_bridge$). See Fig. 6.34 for further visualization of this.

6.7.5.1 Block Dimensions (Layout)

The $RcvrYAddrA$ block dimensions are:

Table 6.24: Subcells used for automation of the Receiver X Address.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
$rcvraddrlatchYA$	$6.6\mu m \times 9.0\mu m$	Cell for latching incoming address.
$validneutral1in4$	$16.8\mu m \times 5.1\mu m$	Valid-Neutral cell for checking that incoming address is valid or neutral state.
$validneutral1in4\_rcvrYA$	$16.8\mu m \times 5.1\mu m$	Valid-Neutral cell for checking that incoming address is valid or neutral state. Used in second CTree.
$RcvrYAddrA\_ctree$	See Above	Second independent CTree with inputs connected to 1-of-4 encoded input for checking that there is a valid address on the input prior to latching onto $rcvraddrlatchYA$.
$Celem\_last$	$16.8\mu m \times 3.4\mu m$	Used in building C-element tree.
$Celem\_single$	$16.8\mu m \times 3.4\mu m$	Used in building C-element tree.
$Celem0$	$16.8\mu m \times 3.4\mu m$	Used in building C-element tree.
$Celem1$	$16.8\mu m \times 3.4\mu m$	Used in building C-element tree.
$Celem2$	$16.8\mu m \times 3.4\mu m$	Used in building C-element tree.
$RcvrXAddr\_bridge$	$16.8\mu m \times 1.0\mu m$	Bridge between $RcvrXAddr$ and $ColumnDecoder$.

$Width = num\_1of4groups \times 16.8\mu m$

$Height \approx 26\mu m$

6.7.5.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

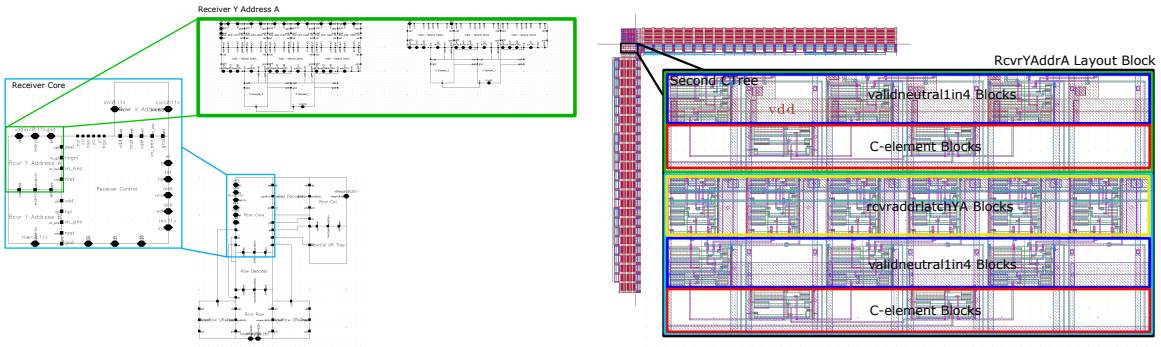
$createRcvrYAddrA\_ctree\_layout.m$: Writes a SKILL function for generating the $RcvrYAddrA_ctree$ layout.

$createRcvrYAddrA\_layout.m$: Writes a SKILL function for generating the $RcvrYAddrA$

layout.

createRcvrYAddrA_schem.m : Writes a SKILL function for generating the *RcvrYAddrA* schematic.

createRcvrYAddrA_symbol.m : Writes a SKILL function for generating the *RcvrYAddrA* symbol.



(a) Schematic (b) Layout

Figure 6.34: Automated layout and schematic of the RcvrYAddrA block.

6.7.6 Receiver Y Address B Automation

The Receiver Y Address B block is responsible for latching the output of the *RcvrYAddrA* block when signaled by the Receiver Control block. It is one of the blocks that make up the complete Receiver Core. It allows for a second row address-event to be latched (onto *RcvrYAddrA*) after the previous has been latched onto this *RcvrYAddrB*. It consists of first *rcvraddrlatchYB* subcells which are directly con-

nected to the incoming address. The number of $rcvraddrlatchYB$ cells is equal to $2 \times num\_1of4groups$. The $num_1of4groups$ value is always the $max(num_1of4groups_row, num_1of4g$ The output of the $rcvraddrlatchYB$ cells is tied to the input of the $validneutral1in4$ subcell. There is one $validneutral1in4$ subcell for every 1-of-4 group. Finally, the C-element tree is placed at the output of the $validneutral1in4$ cells using subcells $Celem\_single$, $Celem\_last$, $Celem0$, $Celem1$ and $Celem2$ (similar to that in the transmitter). There is also exists a filler cells for bridging game between the Receiver Row Decoder and the Receiver Y Address B block ($RcvrYAddrB\_bridge$). See Fig. 6.35 for further visualization of this.

Table 6.25: Subcells used for automation of the Receiver Y Address B.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
$rcvraddrlatchYB$	$8.4\mu m \times 9.2\mu m$	Cell for latching incoming address.
$validneutral1in4$	$16.8\mu m \times 5.1\mu m$	Valid-Neutral cell for checking that incoming address is valid or neutral state.
$Celem\_last$	$16.8\mu m \times 3.4\mu m$	Used in building C-element tree.
$Celem\_single$	$16.8\mu m \times 3.4\mu m$	Used in building C-element tree.
$Celem0$	$16.8\mu m \times 3.4\mu m$	Used in building C-element tree.
$Celem1$	$16.8\mu m \times 3.4\mu m$	Used in building C-element tree.
$Celem2$	$16.8\mu m \times 3.4\mu m$	Used in building C-element tree.
$RcvrYAddrB\_bridge$	$16.8\mu m \times 1.0\mu m$	Bridge between $RcvrYAddrB$ and $RowDecoder$.

6.7.6.1 Block Dimensions (Layout)

The $RcvrYAddrB$ block dimensions are:

$$Width = num\_1of4groups \times 16.8\mu m$$

$Height \approx 17.5\mu m$

6.7.6.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

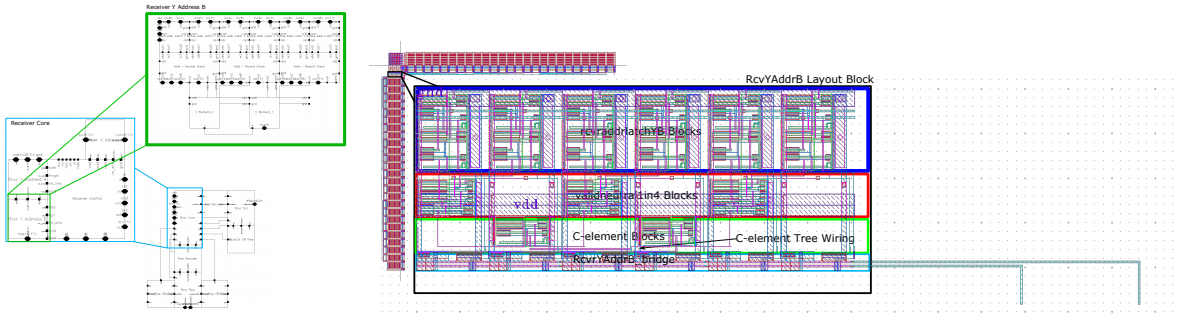
$createRcvrYAddrB\_layout.m$: Writes a SKILL function for generating the $RcvrYAddrB$ layout.

$createRcvrYAddrB\_schem.m$: Writes a SKILL function for generating the $RcvrYAddrB$ schematic.

$createRcvrYAddrB\_symbol.m$: Writes a SKILL function for generating the $RcvrYAddrB$ symbol.

6.7.7 Receiver Control Automation

The Receiver Control block is the main component of the receiver. It is part of the Receiver Core (including also $RcvrYAddrA$, $RcvrYAddrB$, and $RcvrXAddr$). The Receiver Control consists of asynchronous circuits/blocks for controlling receiving row/column addresses and tailword. It controls receiving these address-events and



(a) Schematic (b) Layout

Figure 6.35: Automated layout and schematic of the RcvrY AddrB block.

then latching them so that they can be decoded and sent the appropriate processor array element. It consists of many control signals tied to the supporting blocks in the receiver. The automation for the *rcvrcontrol* only consists of extending metal wires so that they connect to their appropriate signals in supporting blocks. The asynchronous control does not change and is all contained in a single block called *rcvrcontrol*, which never changes regardless of the number of rows and column in the processor array. After the automation of the metal wire extensions, the Receiver Control block is called $rcvrcontrol\_final$, which simply used *rcvrcontrol* and then extends metal connections where necessary. The schematic is always constant regardless the number of rows and columns in the array. This can be seen in Fig. 6.36.

Table 6.26: Subcells used for automation of the Receiver Control block.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
rcvrcontrol	$22.436\mu m \times 18.8\mu m$	Receiver Control block used for automating the complete $rcvrcontrol\_final$ block.

6.7.7.1 Block Dimensions (Layout)

The $rcvrcontrol\_final$ block dimensions are:

$Width = 22.436\mu m$

$Height \approx 49.34\mu m$

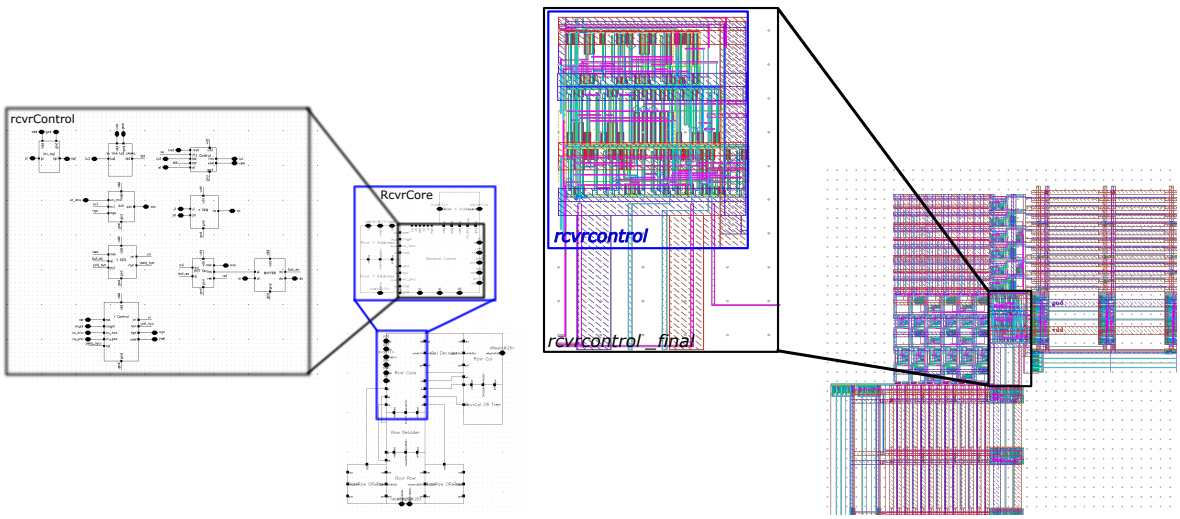
6.7.7.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

$createRcvrControl\_layout.m$: Writes a SKILL function for generating the $rcvrcontrol\_final$ layout.

6.7.8 Receiver Column Automation

The Receiver Column ($RcvrCol$) block is responsible for latching the decoded column request to the processor array. It consists of a single $RcvrColLatch$ subcell for each column. Each subcell is responsible for latching its column request to the



(a) Schematic (b) Layout

Figure 6.36: Automated layout and schematic of the rcvrcontrol_final block.

processor array. The automation also consists of placing these subcells and extending metal wires such that control signals (i.e. rst) are tied to each subcell. This can be seen in Fig. 6.37.

Table 6.27: Subcells used for automation of the Receiver Column block.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
$RcvrColLatch$	$9.1\mu m \times 25.663\mu m$	Cell for latching decoded column request to processor array.

6.7.8.1 Block Dimensions (Layout)

Given num_cols number of columns and a column pitch of $pitch_{horiz}$. The $RcvrCol$ block dimensions are:

$Width = num\_cols \times pitch_{horiz}$

$Height = 25.633\mu m$

6.7.8.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

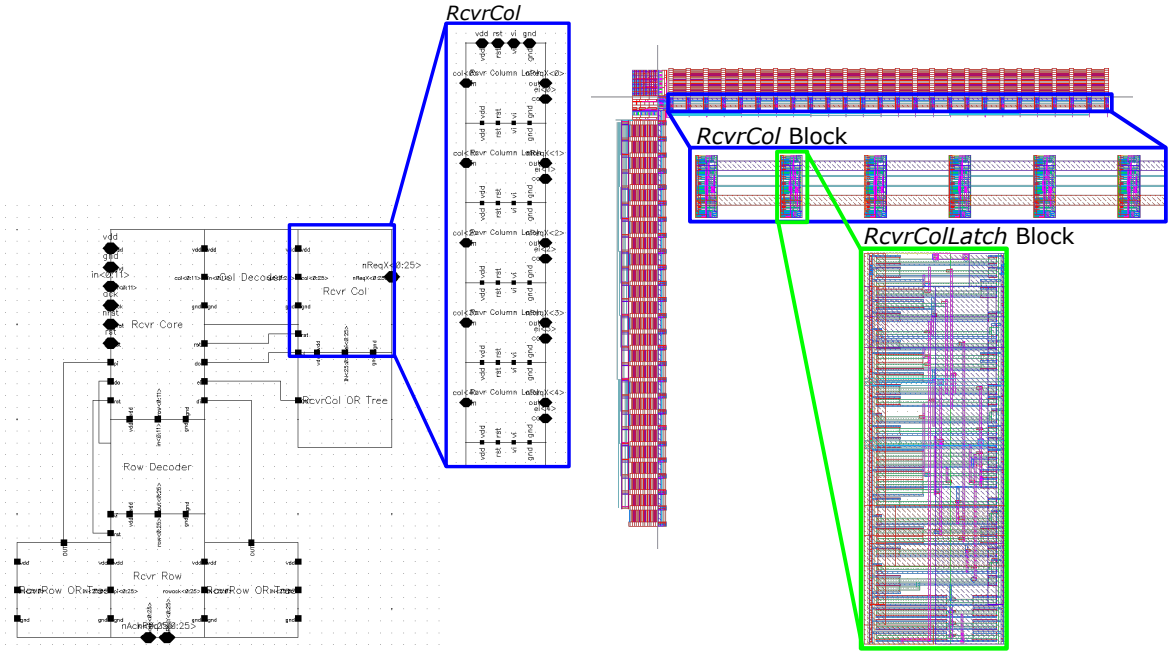
createRcvrCol_layout.m : Writes a SKILL function for generating the *RcvrCol* layout.

createRcvrCol_schem.m : Writes a SKILL function for generating the *RcvrCol* schematic.

createRcvrCol_symbol.m : Writes a SKILL function for generating the *RcvrCol* symbol.

6.7.9 Receiver Row Automation

The Receiver Row (*RcvrRow*) block is responsible for latching the decoded row request to the processor array. It consists of a single *RcvrRowLatch* subcell for each column. Each subcell is responsible for latching its row request to the processor array.



(a) Schematic (b) Layout

Figure 6.37: Automated layout and schematic of the RcvrCol block.

The automation also consists of placing these subcells and extending metal wires such that control signals (i.e. rst) are tied to each subcell. This can be seen in Fig. 6.38.

Table 6.28: Subcells used for automation of the Receiver Row block.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
$RcvrRowLatch$	$18.2\mu m \times 9.1\mu m$	Cell for latching decoded row request to processor array.

6.7.9.1 Block Dimensions (Layout)

Given num_rows number of rows and a row pitch of $pitch_{vert}$. The $RcvrRow$ block dimensions are:

$Width = 18.2\mu m$

$Height = num\_rows \times pitch_{vert}$

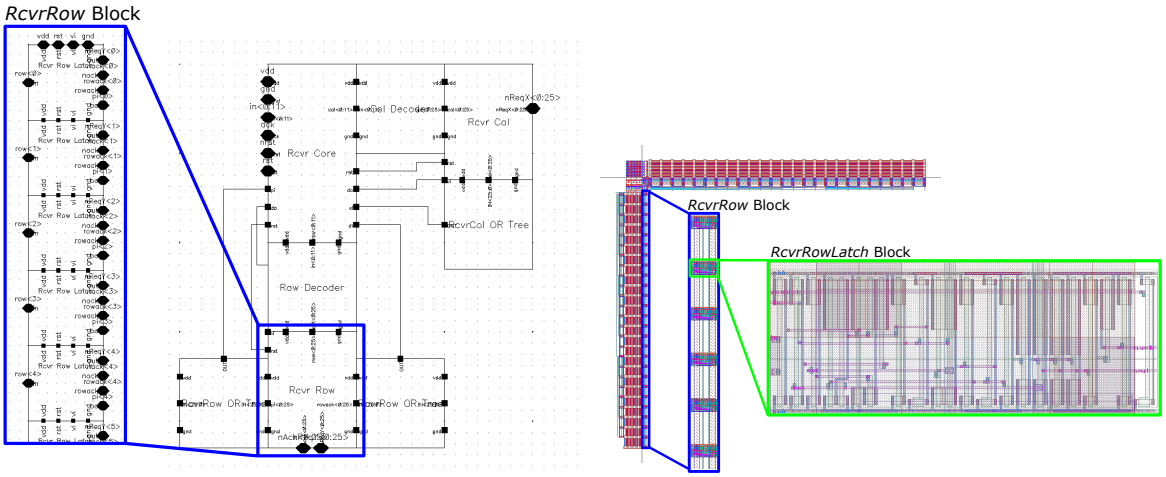
6.7.9.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

$createRcvrRow\_layout.m$: Writes a SKILL function for generating the $RcvrRow$ layout.

createRcvrRow_schem.m : Writes a SKILL function for generating the *RcvrRow* schematic.

createRcvrRow_symbol.m : Writes a SKILL function for generating the *RcvrRow* symbol.



(a) Schematic (b) Layout

Figure 6.38: Automated layout and schematic of the RcvrRow block.

6.7.10 Column Decoder Automation

The Column Decoder is responsible for decoding the 1-of-4 encoded column address into the individual column request lines. It consists of automating each 1-of-4 group decoder individually and then combining them to form the complete Column Decoder (*colDecoder*). Each subdecoder (*Dec_col*1, *Dec_col*2, etc.) consists of *dec* subcells which are essentially nMOS transistors (See Fig. 6.39). The automation in-

volves using the *dec* and *dec_dummy* subcells for creating each individual 1-of-4 group decoder. Then, these subdecoders are combined to form teh final column decoder (See Fig. 6.39). There is also a filler/bridge cell for bridging metal between the Column OR Tree Stack and the Column Decoder. Each subdecoder has dimensions:

$Width = num\_cols \times pitch_{horiz} + 1.886\mu m$

$Height = 16.8\mu m$

Table 6.29: Subcells used for automation of the Receiver Column Decoder block.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
dec	$9.1\mu m \times 4.2\mu m$	Subcell used for creating subdecoders.
dec_dummy	$9.1\mu m \times 4.2\mu m$	Dummy subcell used for creating subdecoders (no transistor in this cell).
dec_end	$9.1\mu m \times 4.2\mu m$	Dummy subcell used for creating subdecoders and placed as last cell.
Dec_col	See Above	Single 1-of-4 group subdecoder.
dec_bridge	$1.886\mu m \times 4.2\mu m$	Bridge subcell placed at each column for bridge between *columnDecoder* and *ORTree_Stack_RcvrCol*.

6.7.10.1 Block Dimensions (Layout)

Given num_cols number of columns and a column pitch of $pitch_{horiz}$. The *colDecoder* block dimensions are:

$Width = num\_cols \times pitch_{horiz} + 1.886\mu m$

$Height = num\_1of4groups \times 16.8\mu m$

6.7.10.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

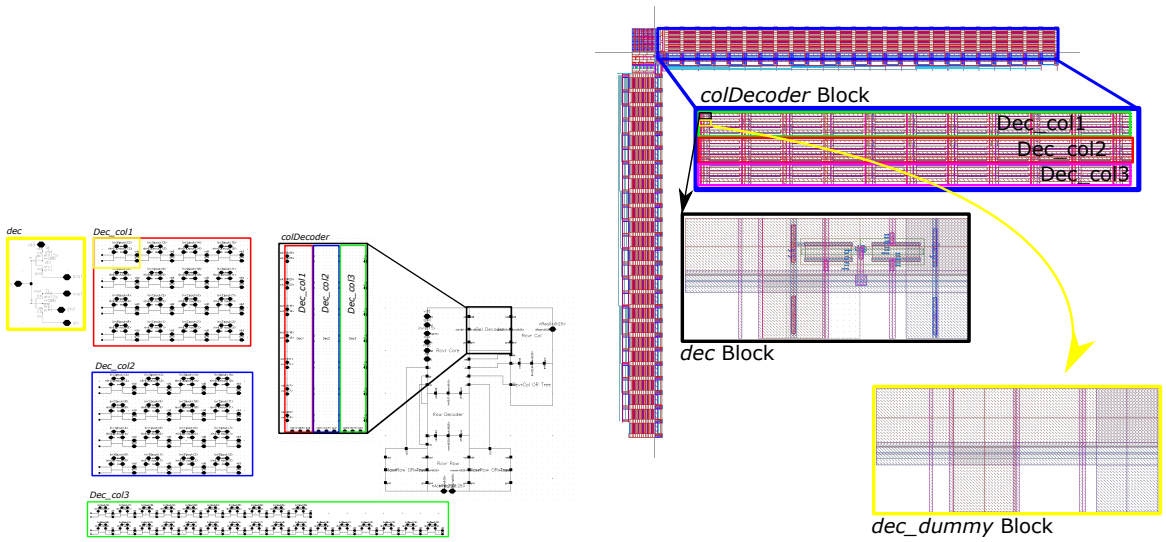
$createRcvrColDecoder\_layout.m$: Writes a SKILL function for generating the $colDecoder$ layout.

$createRcvrColDecoder\_schem.m$: Writes a SKILL function for generating the $colDecoder$ schematic.

$createRcvrColDecoder\_symbol.m$: Writes a SKILL function for generating the $colDecoder$ symbol.

6.7.11 Row Decoder Automation

The Row Decoder is responsible for decoding the 1-of-4 encoded row address into the individual row request lines. It consists of automating each 1-of-4 group decoder individually and then combining them to form the complete Row Decoder ($rowDecoder$). Each subdecoder (Dec_row1, Dec_row2, etc.) consists of dec sub-



(a) Schematic (b) Layout

Figure 6.39: Automated layout and schematic of the colDecoder block.

cells which are essentially nMOS transistors (See Fig. 6.40). The automation involves using the *dec_row* and *dec_row_dummy* subcells for creating each individual 1-of-4 group decoder. Then, these subdecoders are combined to form the final row decoder (See Fig. 6.40). Each subdecoder has dimensions:

$Width = 16.8\mu m$

$Height = num\_rows \times pitch_{vert}$

Table 6.30: Subcells used for automation of the Receiver Row Decoder block.

Subcells Used	**Dimensions ($W \times H$)**	**Description**
dec_row	$9.1\mu m \times 4.2\mu m$	Subcell used for creating subdecoders.
dec_row_dummy	$9.1\mu m \times 4.2\mu m$	Dummy subcell used for creating subdecoders (no transistor in this cell).
dec_row_end	$9.1\mu m \times 4.2\mu m$	Dummy subcell used for creating subdecoders and placed as last cell.
Dec_row	See Above	Single 1-of-4 group subdecoder.

6.7.11.1 Block Dimensions (Layout)

Given num_rows number of rows and a row pitch of $pitch_{vert}$. The $rowDecoder$ block dimensions are:

$$Width = num\_1of4groups \times 16.8\mu m$$

$$Height = num\_rows \times pitch_{vert}$$

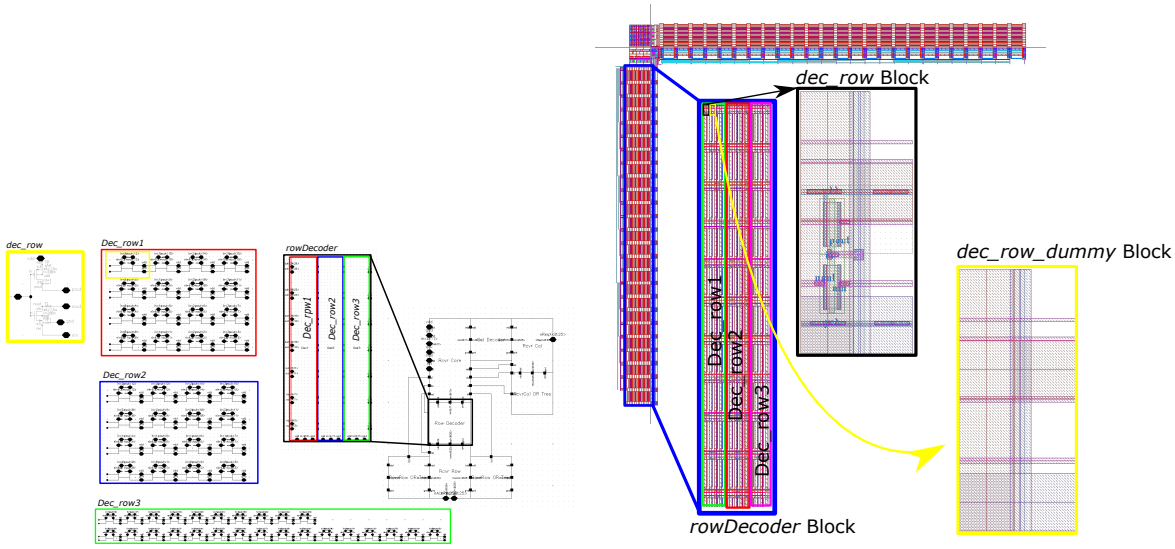
6.7.11.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

$createRcvrRowDecoder\_layout.m$: Writes a SKILL function for generating the $rowDecoder$ layout.

createRcvrRowDecoder_schem.m : Writes a SKILL function for generating the *rowDecoder* schematic.

createRcvrRowDecoder_symbol.m : Writes a SKILL function for generating the *rowDecoder* symbol.



(a) Schematic (b) Layout

Figure 6.40: Automated layout and schematic of the rowDecoder block.

6.7.12 Receiver Core Automation

The Receiver Core is a combination of the *rcvrcontrol_final*, *RcvrXAdd* block, *RcvrYAddrA* block, and *RcverYAddrB* block. These blocks are connected such that it forms the Receiver Core. There is not layout automation for the Receiver

Core, but instead these individual blocks are generated independently. However, the schematic and symbol is automated for the Receiver Core which involves combining these subblocks. The arrangement of these blocks can be seen in Fig. 6.41).

Table 6.31: Subcells used for automation of the Receiver Core block.

Subcells Used	**Dimensions** ($W \times H$)	**Description**
$RcvrYAddrA$	Schematic	Receives input 1-of-4 address and latches row address when signaled by the receiver control.
$RcvrYAddrB$	Schematic	Receives latched 1-of-4 address from $RcvrYAddrA$ and latches row address when signaled by the receiver control.
$RcvrXAddr$	Schematic	Receives 1-of-4 input address and latches column address when signaled by the receiver control.
$rcvrcontrol\_final$	Schematic	Block for controlling receiving of column/row and tailword address and signaling other blocks to latch inputs to outputs.

6.7.12.1 Block Dimensions (Layout)

The *RcvrCore* block consists only of automating the schematic. The layout of the individual blocks that make up the Receiver Core are automated separately and placed appropriately in the full receiver.

6.7.12.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

createRcvrCore_schem.m : Writes a SKILL function for generating the *RcvrCore* schematic.

createRcvrCore_symbol.m : Writes a SKILL function for generating the *RcvrCore* symbol.

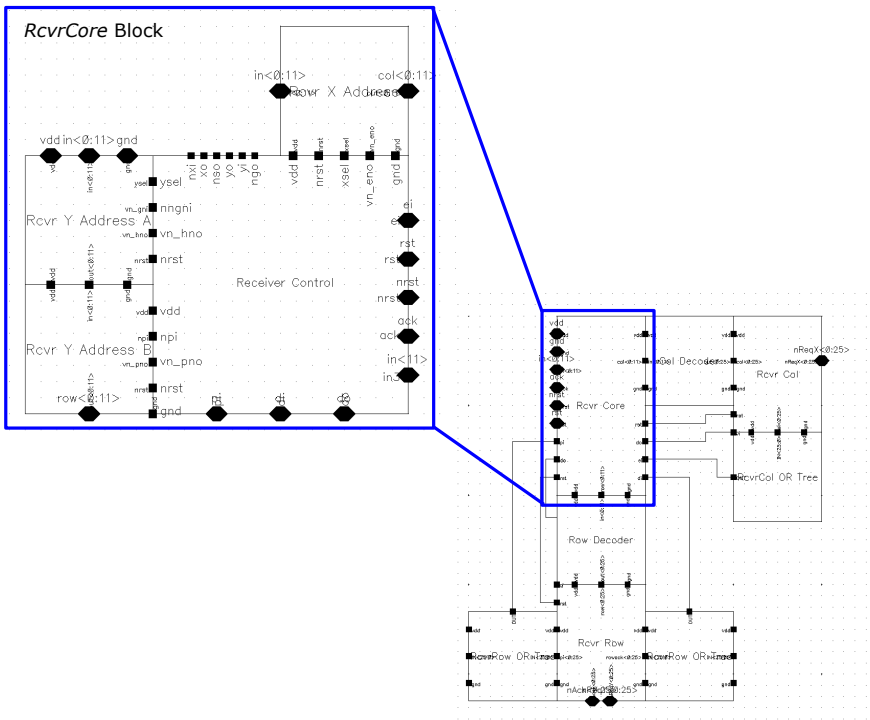


Figure 6.41: Receiver Core schematic automation.

6.7.13 Receiver Input Corner Automation

The Receiver Input Corner block is placed in the top left corner of the receiver. It is the corner block which ties the input 1-of-4 address from the pads to the $RcvrXAddr$ and $RcvrYAddrA$ blocks. It consists of subcell blocks $RcvrInCorner\_mid$, $RcvrInCorner\_left$, and $RcvrInCorner\_right$ for building this complete corner block. The $RcvrInCorner_mid$ block directs the inputs both down and to the right for connection to the $RcvrYAddrA$ and $RcvrXAddr$ blocks, respectively. The $RcvrInCorner\_left$ subcell is a filler on the bottom left side of the complete block for extending the input connection to $RcvrYAddrA$ block. The $RcvrInCorner\_right$ subcell is a filler on the top right of the complete block for extending the input connection to $RcvrXAddr$. Finally, there is a bridge/filler cell placed at the bottom of this $RcvrInCorner$ block. The subcells $RcvrInCorner\_bridge$ and $RcvrInCorner\_bridge\_final$ are used for building this filler/bridge. This can be seen in Fig. 6.42.

Table 6.32: Subcells used for automation of the Receiver Input Corner block.

Subcells Used	**Dimensions** ($W \times H$)	**Description**
$RcvrInCorner\_mid$	$16.8\mu m \times 16.8\mu m$	Bridge for directing input 1-of-4 group down ($RcvrYAddrA$) and to the right ($RcvrXAddr$).
$RcvrInCorner\_left$	$16.8\mu m \times 16.8\mu m$	Bridge for connecting input 1-of-4 group down ($RcvrYAddrA$).
$RcvrInCorner\_right$	$16.8\mu m \times 16.8\mu m$	Bridge for connecting input 1-of-4 group to the right ($RcvrXAddr$).
$RcvrInCorner\_bridge$	$16.8\mu m \times 2.0\mu m$	Bridge/filler between $RcvrInCorner$ and $RcvrYAddrA$.
$RcvrInCorner\_bridge\_final$	$16.8\mu m \times 2.0\mu m$	Bridge/filler between $RcvrInCorner$ and $RcvrYAddrA$ connected to end (bottom left) of block.

6.7.13.1 Block Dimensions (Layout)

The *RcvrRow* block dimensions are:

$Width = num\_1of4groups$

$times 16.8\mu m$

$Height = num\_1of4groups$

$times 16.8\mu m + 2.0\mu m$

6.7.13.2 MATLAB Files

The following MATLAB files are used for performing the computation necessary for generating SKILL Scripts which automate layout, schematic, and symbol of this block:

$createReceiverInputCorner\_layout.m$: Writes a SKILL function for generating the $RcvrInCorner$ layout.

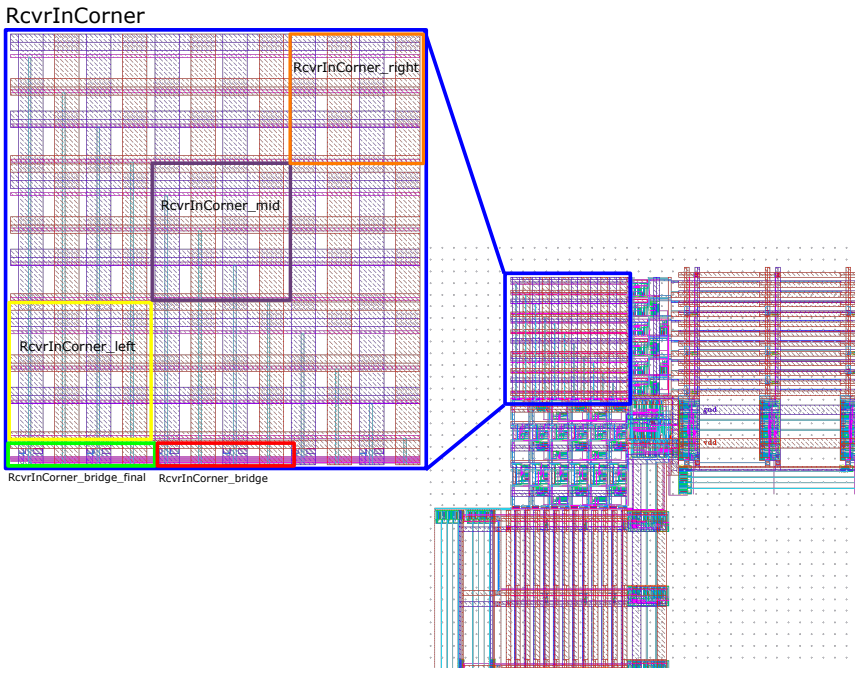


Figure 6.42: Layout of RcvrInCorner block.

6.8 Results and Discussion

6.8.1 Transmitter

6.8.1.1 Area Specifications

Below are the constraints of the transmitter row/column pitch and number of rows/columns in the user's processor array:

Minimum row pitch: $9.1\mu m$

Minimum column pitch: $9.1\mu m$

Minimum number of rows: $1\ rows$

Minimum number of columns: 1 *columns*

The automated transmitter area/dimensions for various array sizes can be seen in Table 6.33. The automated transmitter layout for these 6×6, 26×26, and 46×46 array sizes can be seen in Fig. 6.43.

Table 6.33: Automated transmitter dimensions for various processor array sizes given a vertical/row pitch of $pitch_{vert}$ and a horizontal/column pitch of $pitch_{horiz}$. See Fig. 6.43 for $Bottom_w$, $Bottom_h$, $Right_w$, and $Right_h$ references.

Array Size	Num of Outputs	$Bottom_w$	$Bottom_h$	$Right_w$	$Right_h$
6×6	8	$pitch_{horiz} \times num\_cols + 106.7\mu m$	$126.9\mu m$	$106.7\mu m$	$pitch_{vert} \times num\_rows + 126.9\mu m$
26×26	12	$pitch_{horiz} \times num\_cols + 160.3\mu m$	$191.3\mu m$	$160.3\mu m$	$pitch_{vert} \times num\_rows + 191.3\mu m$
46×46	12	$pitch_{horiz} \times num\_cols + 178.2\mu m$	$213.4\mu m$	$178.2\mu m$	$pitch_{vert} \times num\_rows + 213.4\mu m$

6.8.1.2 Timing Specifications and Simulation

This section includes a table showing timing results for three different transmitters generated for various size processor arrays (6×6, 26×26, and 46×46).

We can determine the transmitter performance by looking at the delay between onset of signals. First, we can determine the delay from when the transmitter receives the row request and when it acknowledges/selects the row. Next we can determine the delay between when the row request is acknowledged/selected and when the

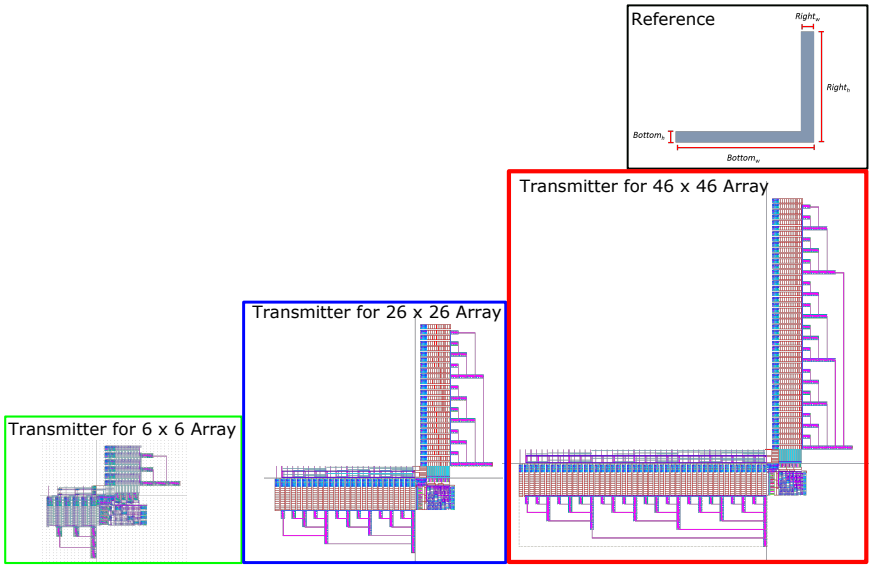


Figure 6.43: Layout of various automated transmitters when different array sizes were inputted (6×6*,* 26×26*, and* 46×46*). The row and column pitch were set to a small value of* $12\mu m$ *for visualization purposes.*

transmitter receives the column request. Next, we determine the delay between the column request to the transmitter and the output of the row address. Next, we determine the delay between the acknowledge received externally for the row address and the output of the column address. And finally, we determine the delay between the acknowledge received externally for the column address and the output of the tailword. The performance of the transmitter can be seen in Table 6.34. This table also shows the total delay assuming no delay between receiving the output address and sending back an acknowledge: $RcvrOffchip_{Delay} = 0ns$.

The timing diagram output of the simulation of the Transmitter can be seen in Fig. 6.44. It shows the timing of the incoming address events (row and column),

Table 6.34: Transmitter performance with respect to the delay between the onset of two signals. The TOTAL DELAY assumes no delay between receiving the output address and sending back an acknowledge.

Start_Signal	**End_Signal**	**Delay(6 × 6)**	**Delay(26 × 26)**	**Delay(46 × 46)**
Xmit Row Address Input Request ($nTReqY < Y >$)	Xmit Row Ack/Select ($TSelY < Y >$)	2.35ns	3.37ns	3.87ns
Xmit Row Ack/Select ($TSelY < Y >$)	Xmit Col Address Input Request ($nReqX < X >$)	0.16ns	0.16ns	0.16ns
Xmit Col Address Input Request ($nReqX < X >$)	Row Address Output (out)	0.53ns	0.53ns	0.53ns
Xmit Acknowledge ($XmitAck$)	Column Address Output (out)	1.16ns	1.21ns	1.31ns
Xmit Acknowledge ($XmitAck$)	Tailword (out)	0.53ns	0.89ns	1.75ns
TOTAL DELAY	-	**4.73ns**	**6.16ns**	**7.62ns**

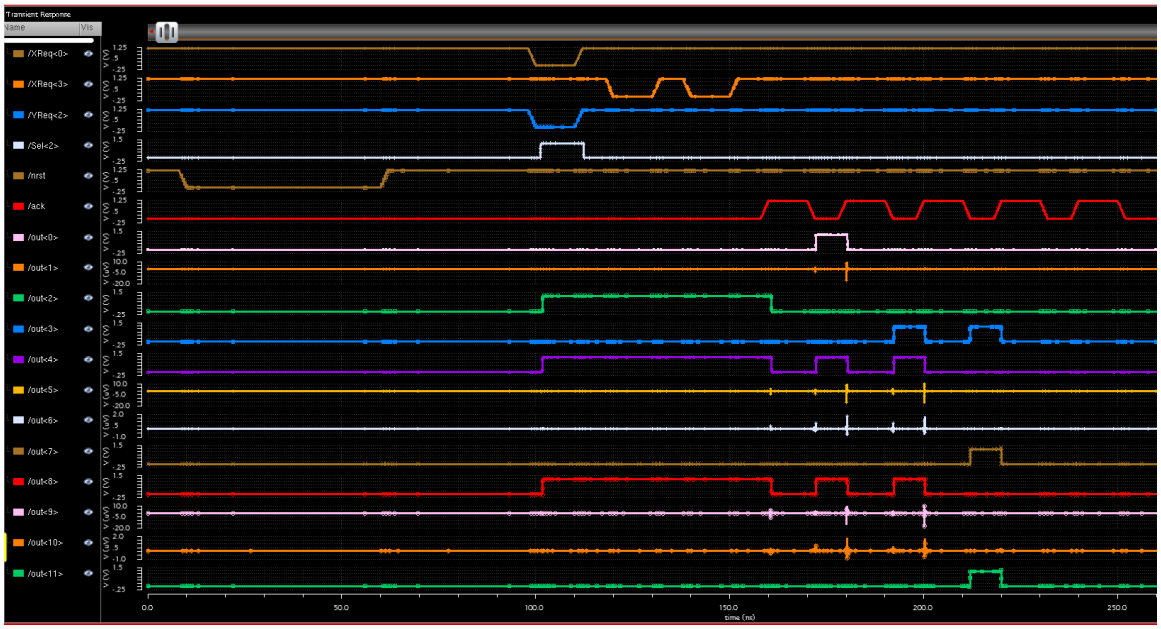


Figure 6.44: Simulation of the transmitter given a 12 × 12 array with row request at row index 2 and column request at column index 0,3, and 3.

the transmitter select/acknowledge signal to the neuron, and the transmit acknowledge signal from the external processor receiving the output address events.

6.8.2 Receiver

6.8.2.1 Area Specifications

Below are the constraints of the receiver row/column pitch and number of rows/columns in the user's processor array:

Minimum row pitch: $9.1\mu m$

Minimum column pitch: $9.1\mu m$

Minimum number of rows: $1\ rows$

Minimum number of columns: $1\ columns$

The automated receiver area/dimensions for various array sizes can be seen in Table 6.35. The automated receiver layout for these 6×6, 26×26, and 46×46 array sizes can be seen in Fig. 6.45.

6.8.2.2 Timing Specifications and Simulation

This section includes a table showing timing results for three different receiver generated for various size processor arrays (6×6, 26×26, and 46×46).

Table 6.35: Automated receiver dimensions for various processor array sizes given a vertical/row pitch of $pitch_{vert}$ and a horizontal/column pitch of $pitch_{horiz}$. See Fig. 6.45 for $Bottom_w$, $Bottom_h$, $Right_w$, and $Right_h$ references.

Array Size	Num of Outputs	$Left_w$	$Left_h$	Top_w	Top_h
6×6	8	$75.0\mu m$	$pitch_{vert} \times num\_rows + 76.1\mu m$	$pitch_{horiz} \times num\_cols + 75.0\mu m$	$76.1\mu m$
26×26	8	$101.43\mu m$	$pitch_{vert} \times num\_rows + 97.33\mu m$	$pitch_{horiz} \times num\_cols + 101.43\mu m$	$97.33\mu m$
46×46	8	$106.29\mu m$	$pitch_{vert} \times num\_rows + 99.74\mu m$	$pitch_{horiz} \times num\_cols + 106.29\mu m$	$99.74\mu m$

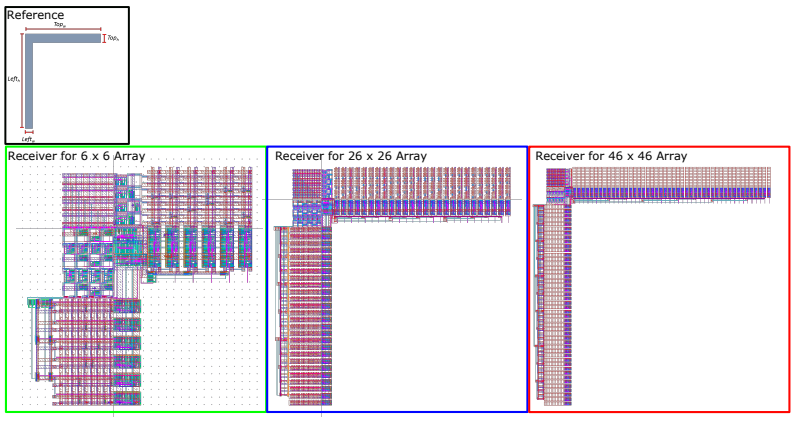


Figure 6.45: Layout of various automated receivers when different array sizes were inputted (6×6, 26×26, and 46×46). The row and column pitch were set to a small value of $12\mu m$ for visualization purposes.

Table 6.36: Receiver performance with respect to the delay between the onset of two signals. The TOTAL DELAY assumes no delay between transmitting the output address and receiving back an acknowledge.

Start_Signal	**End_Signal**	**Delay(6 × 6)**	**Delay(26 × 26)**	**Delay(46 × 46)**
Input Row Address (*in*)	Receiver Acknowledge (*RcvrAck*)	1.13*ns*	1.31*ns*	1.31*ns*
Input Column Address(es) (*in*)	Receiver Acknowledge (*RcvrAck*)	1.12*ns*	1.21*ns*	1.55*ns*
Input Tailword (*in*)	Neuron nReqX and nReqY ($nReqX < X > and nReqY < Y >$)	1.68*ns*	2.01*ns*	2.33*ns*
TOTAL DELAY	-	**3.93ns**	**4.53ns**	**5.19ns**

In this simulation we can determine the performance by looking at delay between onsets of signals. First is the speed at which the receiver acknowledge for row and column are transmitted (after the row and column have been received). Second, the speed at which the receiver output row and column requests are active once the tailword is received. The performance of the receiver can be seen in Table 6.36. This table also shows the total delay assuming no delay between receiving the output address and sending back an acknowledge: $XmitOffchip_{Delay} = 0ns$.

The timing diagram output of the simulation of the Receiver can be seen in Fig. 6.46. It shows the timing of the input address events, receiver acknowledge, output column/row request, and neuron acknowledge.

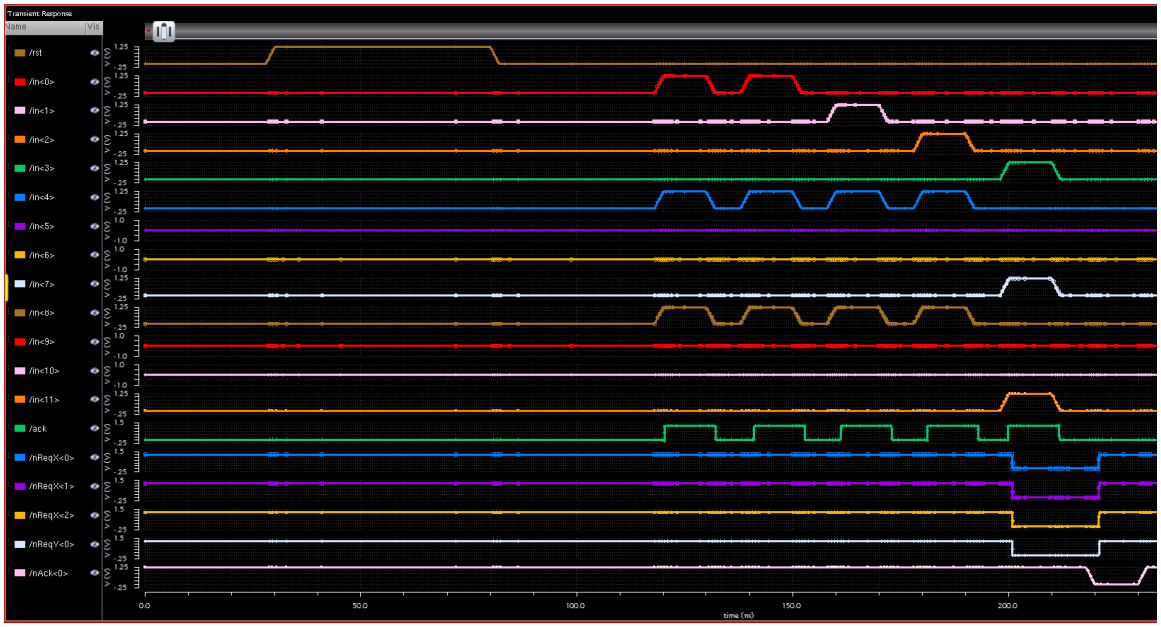


Figure 6.46: Simulation of the receiver given a 12×12 *array with an event at row index 0 and column index 0,1, and 2.*

6.8.3 Full Chip

6.8.3.1 Timing

The max event rate is limited by the time a neuron receives an event to the time it generates an event. This delay is ∼50ns. Therefore, the total time for sending an event and receiving an event is dependent on the size of the array. Fig. 6.47 depicts the maximum event rate as a function of $N \times N$ array size. For this 12×12 array, the maximum event rate ∼18.5 Mevents/second.

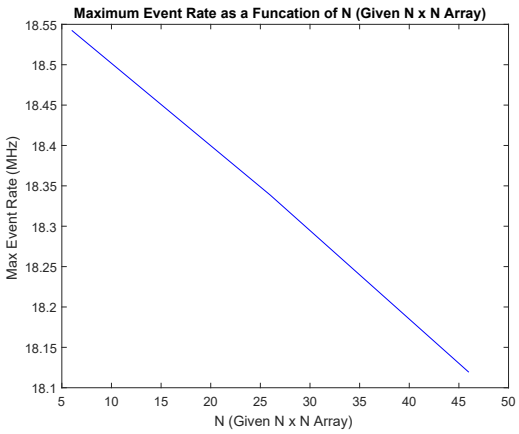


Figure 6.47: Maximum event rate as a function of N (such that the array size is $N \times N$).

6.8.3.2 Simulation

Simulation results can be seen in Fig. 6.48. For simulating the full chip, we send events at a rate of 10 KHz to neuron (1,1). The simulations validate proper operation of the full chip.

6.8.4 Array Characterization

To characterize the neurons in the array, we looked at the following: mismatch between neurons, average output event rate as a function of input event rate, effect of varying the threshold voltage (Vth), effect of varying the weight (W), and the effect of varying the synaptic driving potential (E).

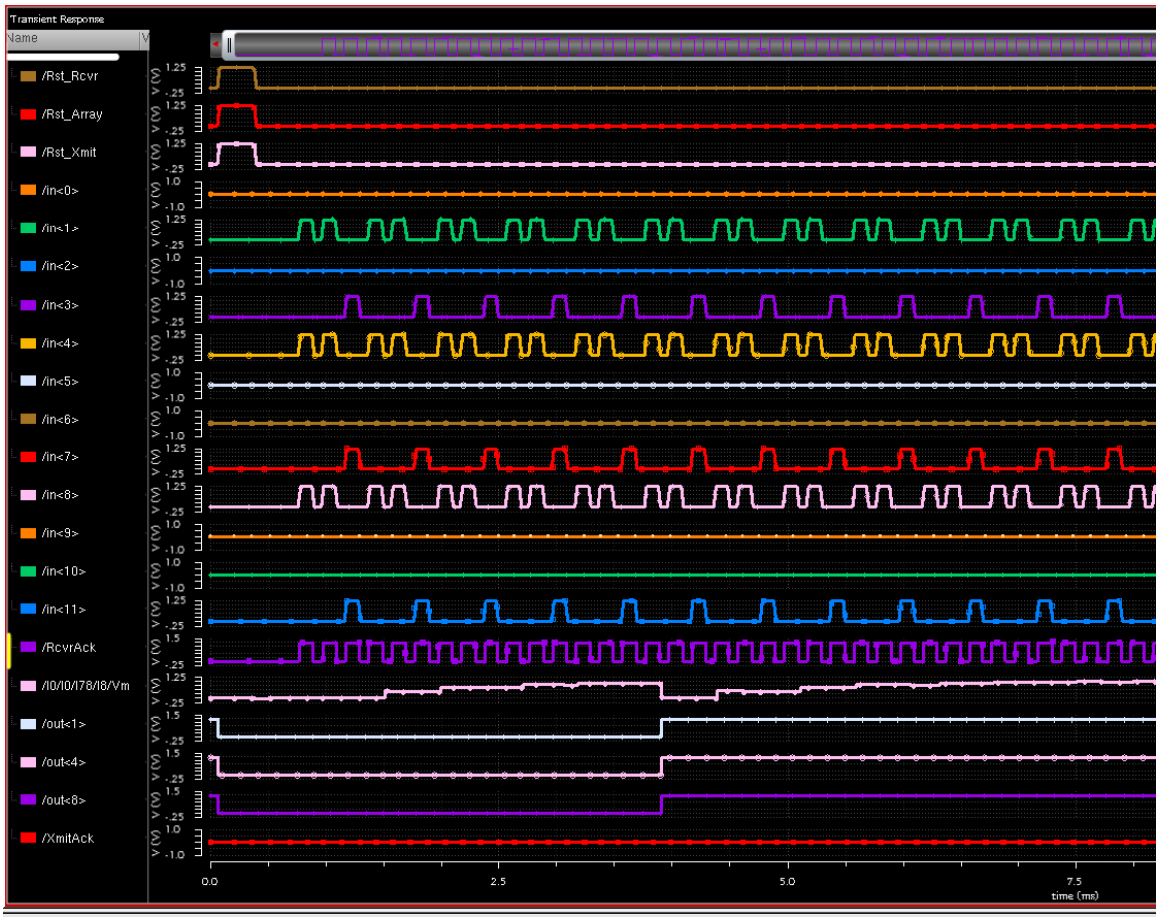


Figure 6.48: Simulation results for the 12×12 *array full chip. Events are sent to neuron (1,1) at an event rate of 10 KHz.*

6.8.4.1 Mismatch

Considering each neuron had an independent circuitry including comparator (soma), synapse, and membrane capacitance, there was considerable mismatch between neurons in the array. To characterize the mismatch, we set $W = 2$, $E = 1.10V$, and $Vth = 0.75V$. We then send events at a rate of ∼25 KHz. The mean of the distribution of output events to input events ratio is around 0.2. This can be seen in Fig. 6.49 and is also visualized in the image on the right.

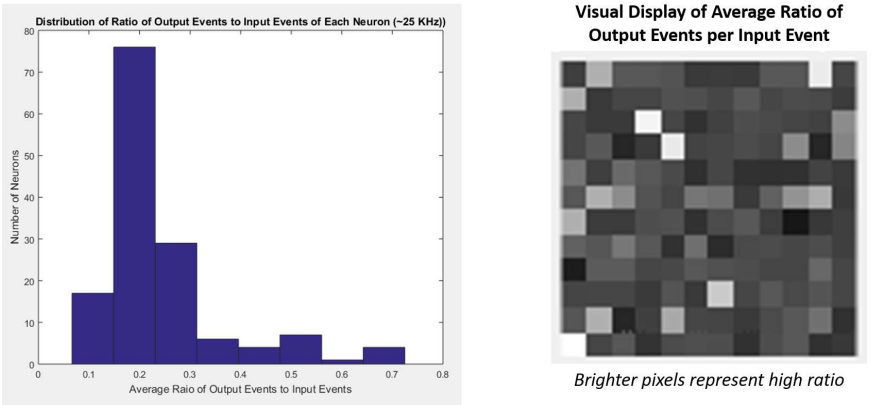


Figure 6.49: Mismatch Characterization of 12×12 *IFAT in 55nm technology.*

6.8.4.2 Varying Input Event Rate

The effect of varying the input event rate was characterized. The input event rate was varied between 0 Hz and 1.1 MHz in intervals of 100 KHz. When varying the input event rate, we set $W = 1$, $E = 1.10V$, and $Vth = 0.75V$. The change in output event rate with respect to input event rate was linear (See Fig. 6.50).

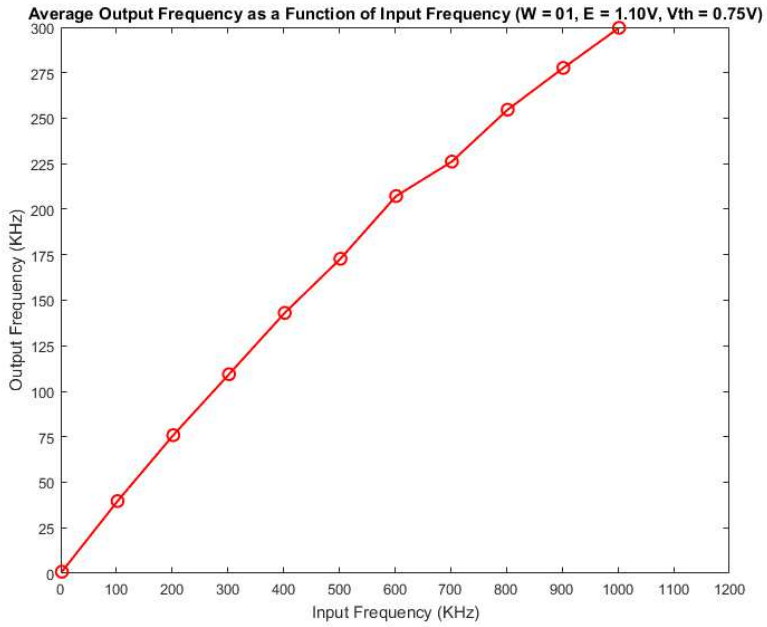


Figure 6.50: Effect of varying input event rate and observing average output event rate.

6.8.4.3 Varying Synaptic Driving Potential (E)

The weight of the synaptic connection between two neurons is dependent on the parameter E, which is the synaptic driving potential. To characterize the effect of the synaptic driving potential, we set $Vth = 0.55V$ and we send events at a rate of 1.0 MHz and observe the average ratio of output events to input events for both $W = 1$ and $W = 3$. The result is an increasing linear relationship as E increases. This linear relationship is further shifted upward for a greater weight ($W = 3$). This can be seen in Fig. 6.51.

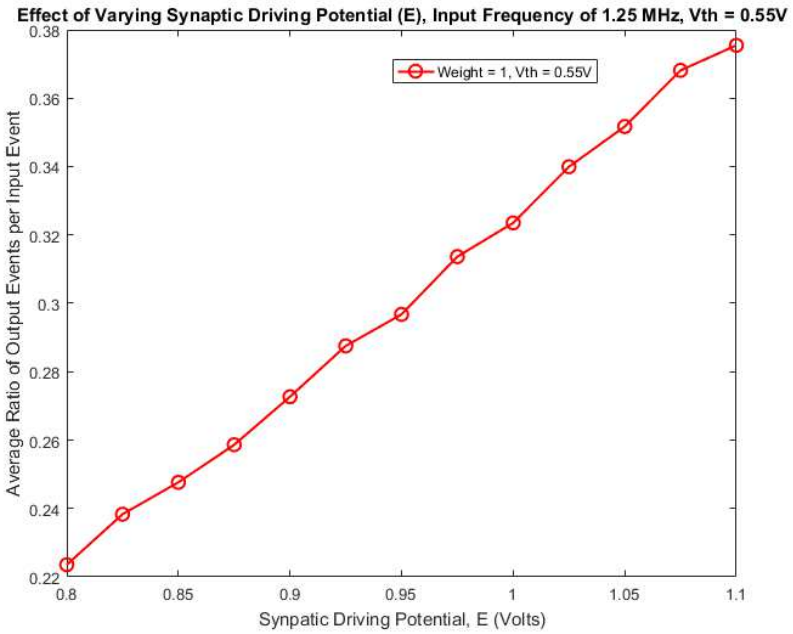


Figure 6.51: Effect of varying synaptic driving potential and observing average output event to input event ratio.

6.8.4.4 Varying Synaptic Weight (W)

The weight of the synaptic connection between two neurons is also a function of the weight W which determines the size of the synapse capacitance. The greater W, the greater the synapse capacitance, and therefore, the more charge integrated onto the membrane capacitance for a given event. To characterize the effect of varying the weight, W, we varied W between 0 and 3 considering it is only 2 bits in precision. This experiment was performed for $E = 0.8V$, $E = 1.0V$ and $E = 1.2V$. Events were sent an a rate of 1.0 MHz and the average output event to input event ratio was observed. As seen in Fig. 6.52, there is a linear relationship and this linearity is shifted upward as E increases.

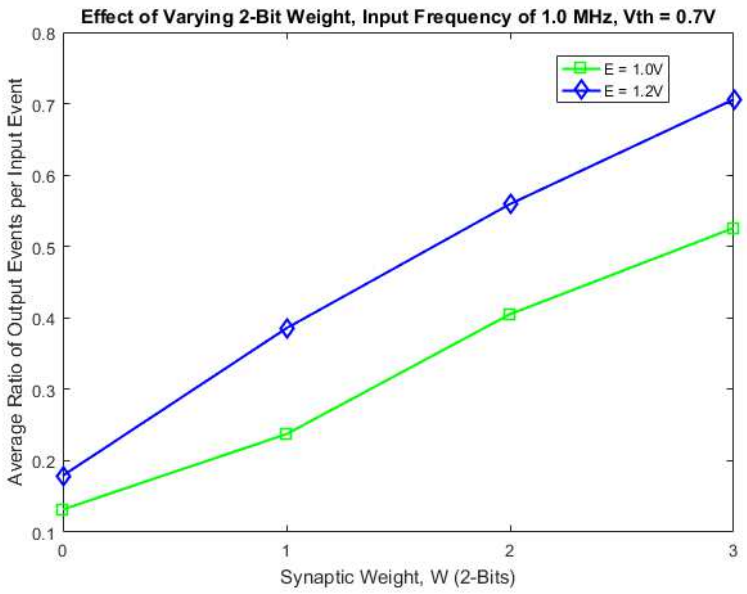


Figure 6.52: Effect of varying synaptic weight and observing average output event to input event ratio.

6.8.4.5 Varying Threshold Voltage (Vth)

To characterize the effect of varying the threshold voltage, we set the input event rate to 1.0 MHz. We set $E = 1.15V$ and observe the average output event to input event ratio for $W = 1$ and $W = 3$. There is a decreasing linear relationship as the threshold voltage (Vth) increases. This is due to the fact that a higher membrane voltage is required for reaching the voltage and therefore more input events are needed to exceed the threshold voltage as it increases. This linear relationship is shifted upwards as the weight, W, increases (See Fig. 6.53).

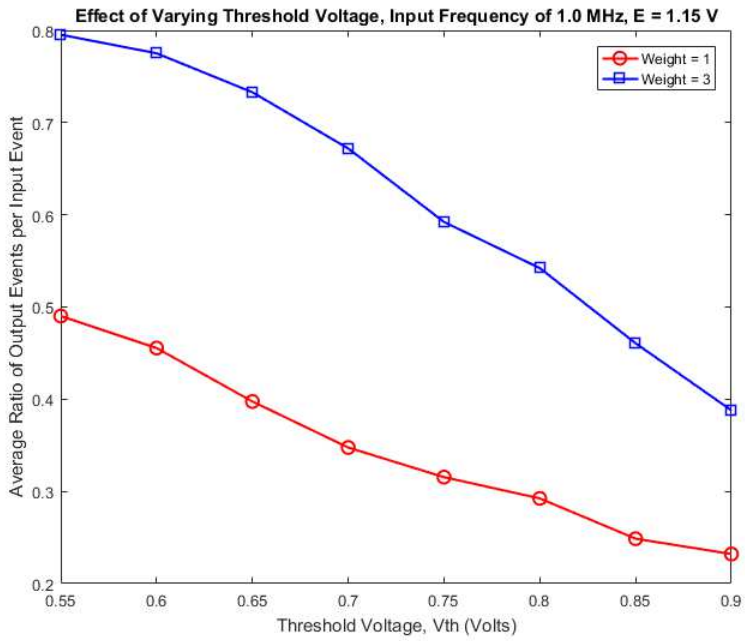


Figure 6.53: Effect of varying threshold voltage and observing average output event to input event ratio.

6.9 Visual Processing Application

To validate this system, we apply it to a visual pre-processing task. Specifically, we see to utilize the system to perform two tasks:

1. Dewarping

2. Filtering

The dynamic reconfigurability of the IFAT system allows for these two tasks, dewarping and filtering, to be performed simultaneously and in an event-based approach. Such a task is essential for any mobile system performing a visual processing task. When camera motion exists, dewarping of the camera output is necessary for accurate computation of proceeding visual tasks, such as in this case, object tracking. We demonstrate a simulation results of the IFAT system in doing a simultaneous dewarping and filtering task, specifically a debayering task. We then demonstrate experimental results of the 12×12 system in performing a dewarping task, filtering task, and simultaneous dewarping and filtering task.

6.9.1 Simulation

Debayering is necessary when the camera output is a bayered image. A bayered image is result of a bayer filter, which is a color filter array arranging red (R), green (G), and blue (B) color filters on a square grid of pixels. This can be seen in Fig. 6.54.

The bayered image is a grayscaled image in which each pixel intensity is result of one of the R, G, or B color filters. The filter pattern is typically 25% red, 50% green, and 25% blue. We utilize a software emulation of the IFAT system to demonstrate this debayering task.

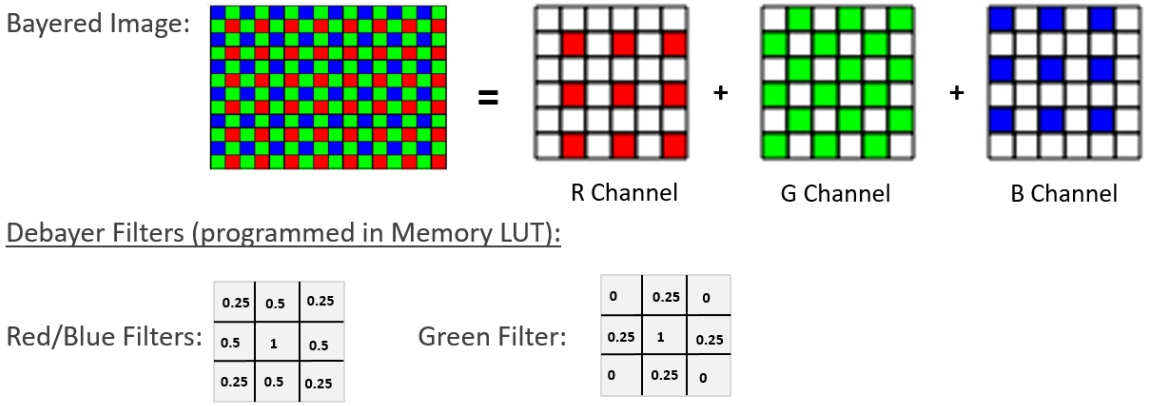


Figure 6.54: Visual explanation of the debayering and filters applied to generate R,G, and B channels.

The flow of the system can be seen in Fig. 6.55. However, instead of using a single IFAT array, we use three different arrays for representing the red, blue, and green channels. The filters seen in Fig. 6.54 are used for each of the red, green, and blue channel LUTs accordingly such that the proper spreading operation is performed within each channel. The camera is represented by the blue and red square, in which the red side represents the top of the camera. We assume the output is a bayered image. The events generated by the pixel intensities of this bayered image are propogated to the LUT of each of the R, G, and B channels. The camera is rotating and translating, and therefore, we assume the camera motion is known, and

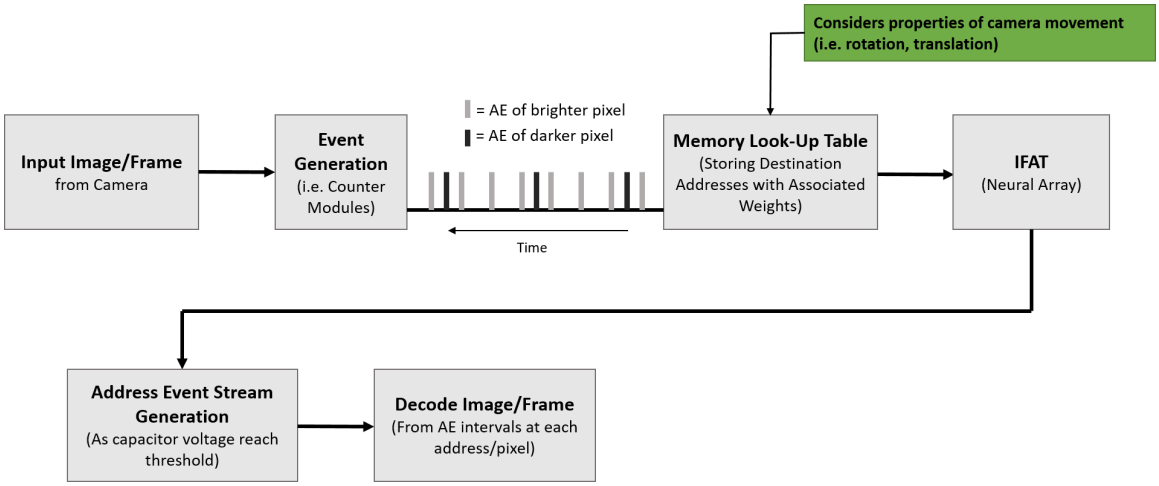


Figure 6.55: Flow of the IFAT software emulation for validating its capability of performing visual tasks.

we apply a rotation matrix to determine the destination addresses of the incoming address events generated from the camera. The same destination locations for each of the incoming events are identical for each of the three arrays (within R, G, and B channels). However, the associated synaptic weights are different for the green channel, and the same for the red and blue channels considering the kernel is different for performing the spreading operation. The output spike train from each of the three arrays are decoded and represented as pixel intensities within their appropriate channel. We finally sum these three channels to achieve the final result. The decoded RGB color image represents the original RGB image over the areas that the camera has explored, and with high accuracy. A screenshot of this system can be seen in Fig. 6.56. Average error per frame (over time) in regards to difference in pixel intensity to the IFAT system computed RGB image and the original MATLAB RGB image

can be seen in Fig. 6.57.

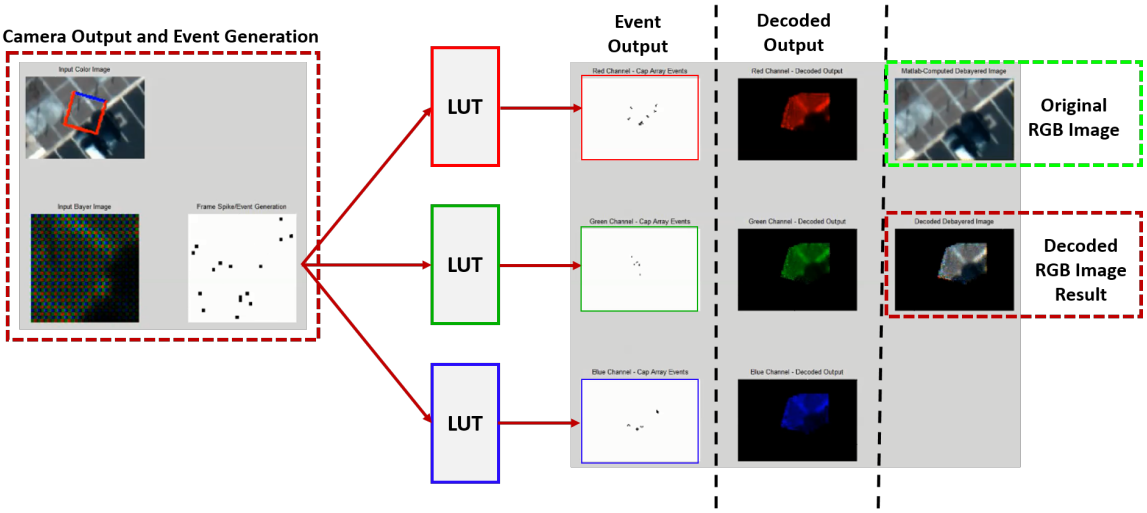


Figure 6.56: Screenshot of the debayering task applied to a three IFAT software emulation.

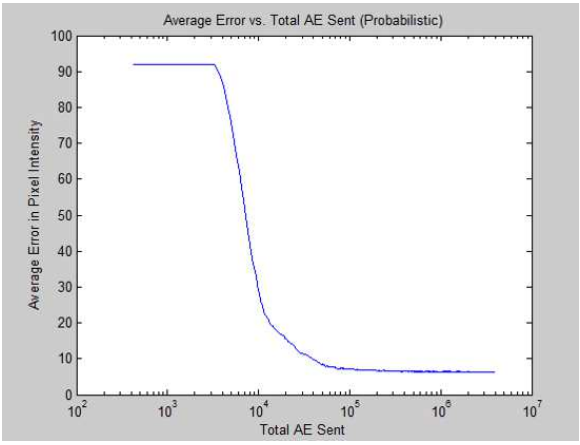


Figure 6.57: Average error (pixel intensity difference) of the IFAT system-computed RGB image against the original MATLAB RGB image (of the region the camera has explored). Error is displayed as a function of number of events generated (essentially time).

Figure 6.58: Printed circuit board containing the IFAT chip designed in 55nm technology and Opal Kelly 310 FPGA (Spartan-3).

6.9.2 Experimental

Considering the chip consists of a smaller array of 12×12 resolution, we do not show a complete debayering task, but rather demonstrate its ability to decode a small 12×12 image with one-to-one mapping, filtering, and simultaneous dewarping and filtering. We interface the IFAT chip with an Opal Kelly 3010 FPGA board containing a Spartan-3 FPGA. The complete PCB containing the chip and FPGA can be seen in Fig. 6.58. The process for controlling the chip and performing visual tasks can be seen in Fig. 6.59.

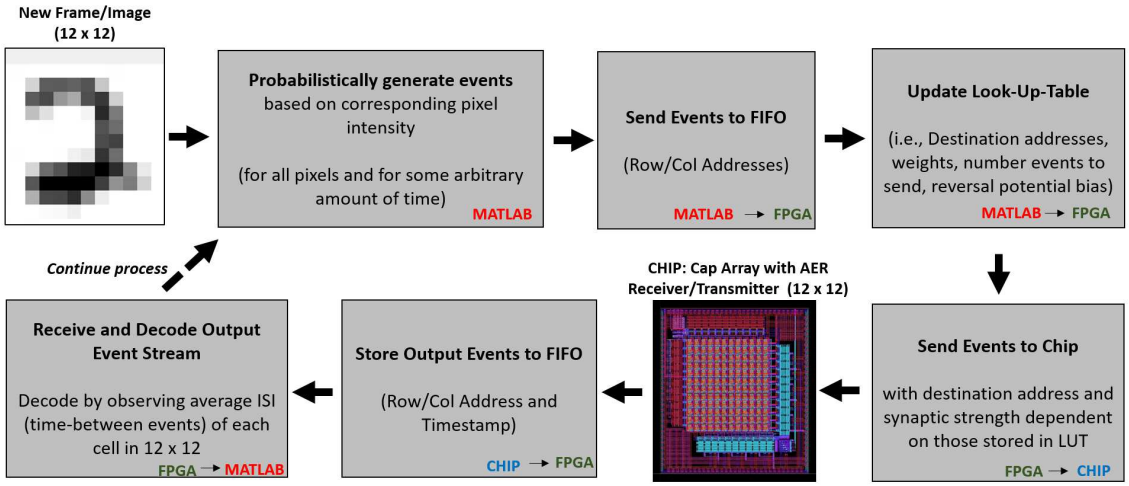


Figure 6.59: Flow of the visual tasks performed via the IFAT system.

Address events are generated via MATLAB probabilistically from each pixel with a mean firing rate proportional to the pixel intensity. The events are sent to a FIFO on the FPGA via USB 2.0 communication. The LUT on the FPGA (using Block RAM) is programmed with the appropriate destination addresses and weights. Address events

received by the FPGA go through the LUT which determine the destination addresses to which events are then sent on the IFAT chip. As neurons output events from the chip, the output events are stored in an additional FIFO. These output address events are transmitted to the PC/MATLAB via the same USB 2.0 communication. Each address event is tagged with a timestamp representing the time at which the neuron generated the event.

6.9.2.1 One-to-one Mapping

The first experimental test was to confirm proper decoding of the image basted on the spike rate of each neuron. For this we setup a one-to-one mapping within the LUT such that each event generated by a pixel was projected to the same address in the neuron array as that of the pixel location. The result can be seen in Fig. 6.60.

6.9.2.2 Filtering/Smoothing

The second experimental test was to perform a vertical smoothing task using the kernel seen in Fig. 6.61. Address events were not only projected to the same location in the neuron array, but also to the neurons above and below with a weight of $\frac{1}{3}$. This resulted in a vertical smoothing operation which can be seen in Fig. 6.61.

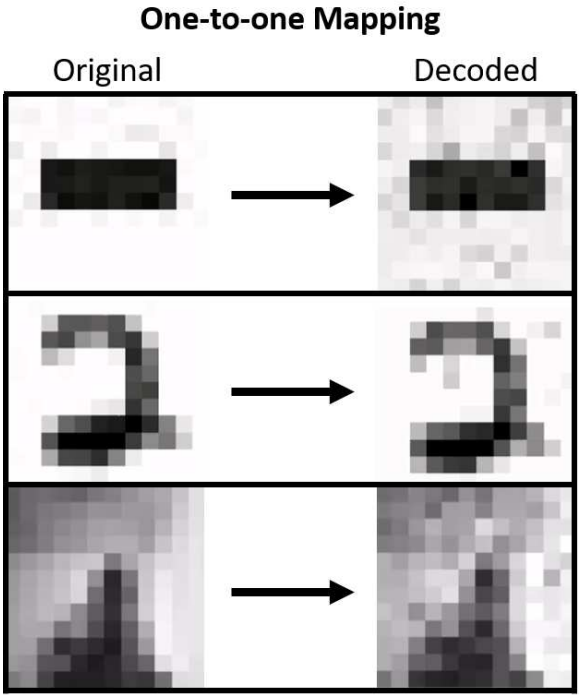


Figure 6.60: Experimental results of one-to-one mapping using IFAT chip.

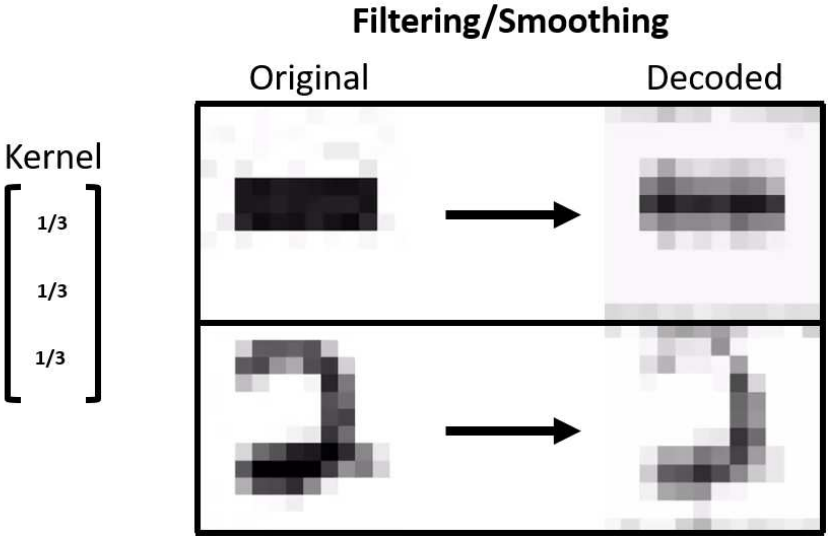


Figure 6.61: Experimental results of smoothing operation using IFAT chip.

6.9.2.3 Dewarping

Finally, we perform a dewarping task given rotation about the center of the image. Given the input image is rotating, the rotation is known, and therefore, the LUT is programmed such that incoming address events are dewarped to their appropriate location to keep the image fixed about a common coordinate system. The top result in Fig. 6.62 is the result of this dewarping task at a particular point in time at which the horizontal bar has rotated about $\sim 45^{\circ}$ clockwise. The bottom result demonstrates the ability to perform this same dewarping task while simultaneously performing the smoothing task previously shown.

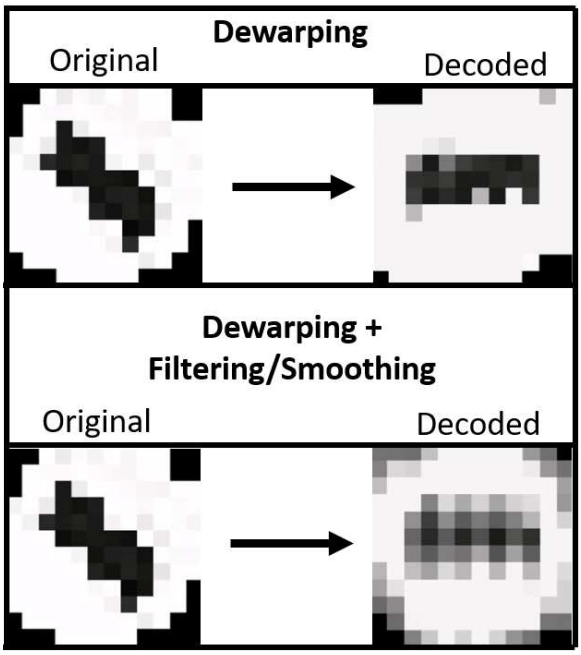


Figure 6.62: Experimental results of a dewarping task as well as a simultaneous dewarping and smoothing task.

6.10 Conclusion

In conclusion we have demonstrated an IFAT system implemented in 55nm technology using an I&F neuron model with conductance-based synapses. A key novelty to this system is the automated delay-insensitive AER receiver and transmitter design for any system which takes advantage of the AER communication protocol. To our knowledge, this is the only system which can automatically generate both the receiver and transmitter layout, schematic, and symbol (LVS-pass and DRC-free). With entering only the number of rows, number of columns, and row and column pitch of the event-based array, the entire AER receiver and transmitter can be generated within seconds. This results in tremendous savings in design time. Traditional digital synthesis software cannot be used because the AER system is not purely digital but rather uses asynchronous circuits. We further demonstrate the ability to perform visual pre-processing tasks including one-to-one mapping, filtering (smoothing), dewarping, and simultaneous dewarping and smoothing tasks.

This work was conducted with Dr. Andreas Andreou's lab. Furthermore, this work as been demoed to DoD DARPA and served as a basis for the both the UPSIDE and ReImagine DoD DARPA grants won (>$1 Million in funding).

Chapter 7

Conclusion and Future Work

7.1 Future Work

In the following sections we will discuss where the future of this work lies and near-future objectives.

7.1.1 Saliency in Virtual Reality

One future project includes the application of saliency within a virtual reality environment (See Fig. 7.1). Specifically, we seek to integrate with the Oculus Rift. The Oculus Rift is a head-mounted device used for a virtual reality display. The first development kit version of the Oculus Rift was released in 2012 and the second version in 2014. We seek to utilize the Oculus Rift DK2 for applying the saliency model. Current work involves working the the Oculus to allow for full control of displaying

360 images and video within the Oculus Rift display. Such control will allow for integration of the saliency model with the Oculus Rift virtual reality environment. We seek to apply the saliency in two ways:

1. "Augmented Vision": The first consists of computing saliency in real-time at only the field of view of the viewer. Therefore, as the user navigates within the virtual reality world, saliency will be computed in real-time at the location the viewer is facing. By utilizing the saliency model in this manner, it acts as augmented vision. The saliency map will quickly tell the user the most salient, interesting regions within their field of view. The FPGA implementation of the model will be used for this real-time processing.

2. "Where-to-Look": The second manner in which we seek to utilize the saliency model with the Oculus Rift is by computing the saliency map on the entire 360 image in real-time. By doing so, the salient regions of the entire virtual world will act as target locations for telling the viewer where-to-look. It will notify the viewer where the salient locations are in real-time within the entire 360 degree image. This will require computing the saliency map on the entire 360 degree image in real-time. The FPGA implementation will also be used for such real-time processing.

Such work will require integration of an eye tracker within the VR display. The eye tracker within the Oculus Rift will consists of a small infrared (IR) camera, an IR

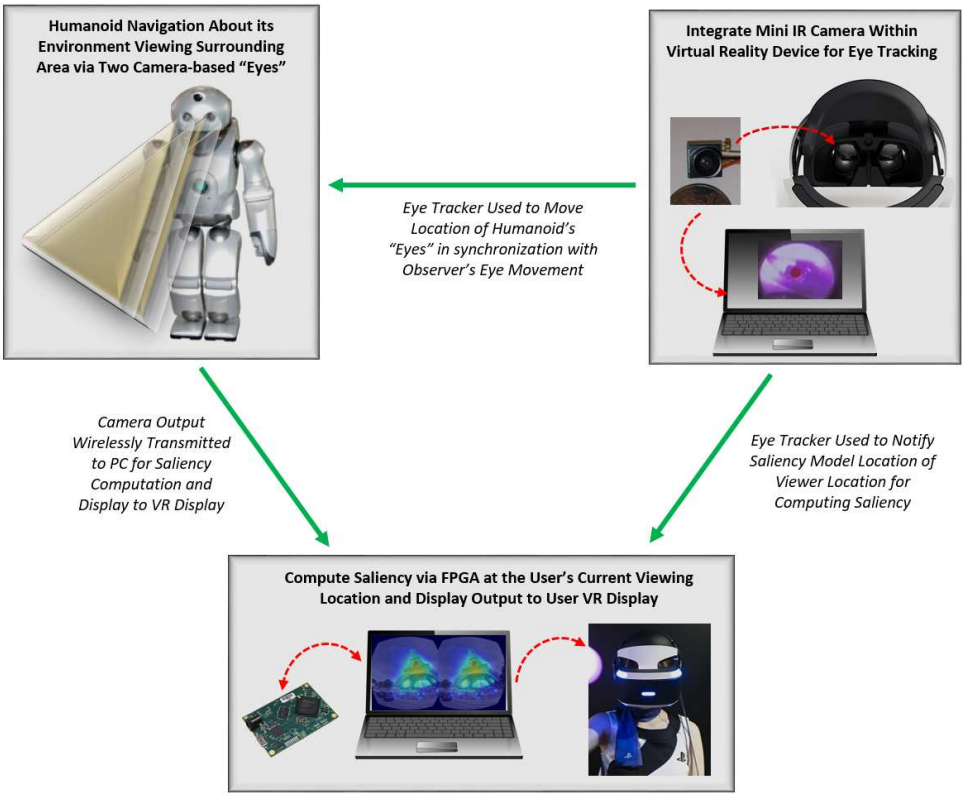


Figure 7.1: Saliency within VR system for Augmented Vision or "Where-to-Look" applications in a virtual reality environment.

("hot") mirror (for reflecting pupil motion to the IR camera), IR LEDs to illuminate the eye, and a TS832 transmitter to send the IR camera output to the RC832 receiver connected to the laptop. The laptop contains the eye tracking software, which uses the output of the IR camera within the Oculus Rift as input. Two additional transmitter/receiver pairs are used for the two FPV cameras on the mobile robot (e.g. humanoid) for sending video of what the robot "sees" to the laptop. This stereo video from the robot is displayed to the Oculus Rift. An Arduino is mounted onto the RC car, controlling the motion of the robot as well as the two cameras attached to the robot. Eye tracking will be used for determining where saliency should be computed within the entire virtual environment. It will also be used for controlling motion of the two FPV cameras on the robot. Head motion will be used for controlling motion of the car itself. Head position data is readily available from the Oculus Rift.

Recently, there is interest in the application of saliency in a VR environment [158, 159].

7.1.2 Self-contained IFAT

One of the bottlenecks in the current IFAT implementation is the time required to access the LUT and set the synaptic weight parameters (switch-cap synapse weight (W) and synaptic driving potential (Em)) each time an event is received by the IFAT system. This limits the maximum event rate at which events can be sent to the IFAT system and also causes reliance on the speed of the FPGA clock. To mitigate this

delay, we propose an IFAT system in which the LUT memory (SRAM) and digital-to-analog converter (DAC) are on-chip. The neural array itself will not change, there will only be additional circuitry for these components. This system will be designed in 130nm technology and if the neuron size scales with feature size, it should allow $\sim 10,000$ neurons/mm^2. This self-contained IFAT can visualization can be seen in Fig. 7.2. This is similar to the small array of 32 neurons designed by Indiveri et al. [160], however, their design did not include an on-hip DAC for setting analog biases. This novel analog system can facilitate off-chip control and mitigate delays due to memory access and bias setting.

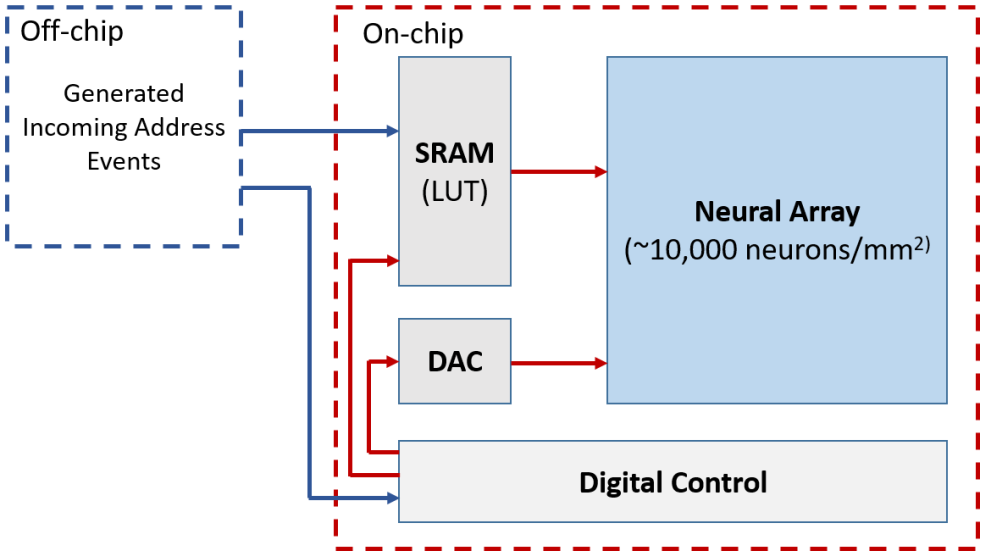


Figure 7.2: Self-contained IFAT design with SRAM-based LUT and DAC on-chip.

7.1.3 Multi-Chip IFAT System for Visual Saliency Computation

Another near-future objective is to design a PCB with at least four IFAT chips (discussed in Chapter 5 capable of working in parallel. This multiplies the number of available neurons by a factor of 4. Given the chip depicted in Chapter 5, such a 4-chip system will allow for 16,320 I&F neurons and 8,160 Mihalas-Niebur neurons. The idea of this 4-chip system is to be a generic IFAT system that can be configured to achieve any event-based tasks (visual task, learning task, etc.). Specifically, we seek to use this system to perform the convolutions necessary for the visual saliency model. This will allow for a low-power event-based implementation of the visual saliency model in VLSI technology. This system can be coupled with a drone for performing such visual tasks in real-time. This system is visualized in Fig. 7.3. This will be similar to various other large-scale neural networks implemented in VLSI technology which have integrated multiple chips (or cores) onto a single board ([28], [27], [26], [12], [133]). In this system, there will be single global FPGA which will interface with 4 smaller FPGAs, one coupled to each of the 4 IFAT chips. Each of the local FPGAs will serve two purposes. It will control the sending and receiving of events to and from the chip, and further, to and from the global FPGA. Secondly, it will utilize block RAM for serving as the LUT for the IFAT chip it is coupled to. This system will be general-purpose and be capable of being reprogrammed for any application. With respect to

the visual saliency model, considering the high number of convolutions necessary for the border ownership and grouping activity computation, this task can be off-loaded to the IFAT. These convolutions are essentially edge and center-surround filters which can be computed via the IFAT. We seek take this one step further and couple an ATIS with IMU to to the IFAT system for the capability of performing the dewarping task and visual saliency task simultaneously with known, synchronized camera motion data. This idea is further visualized in Fig. 7.3.

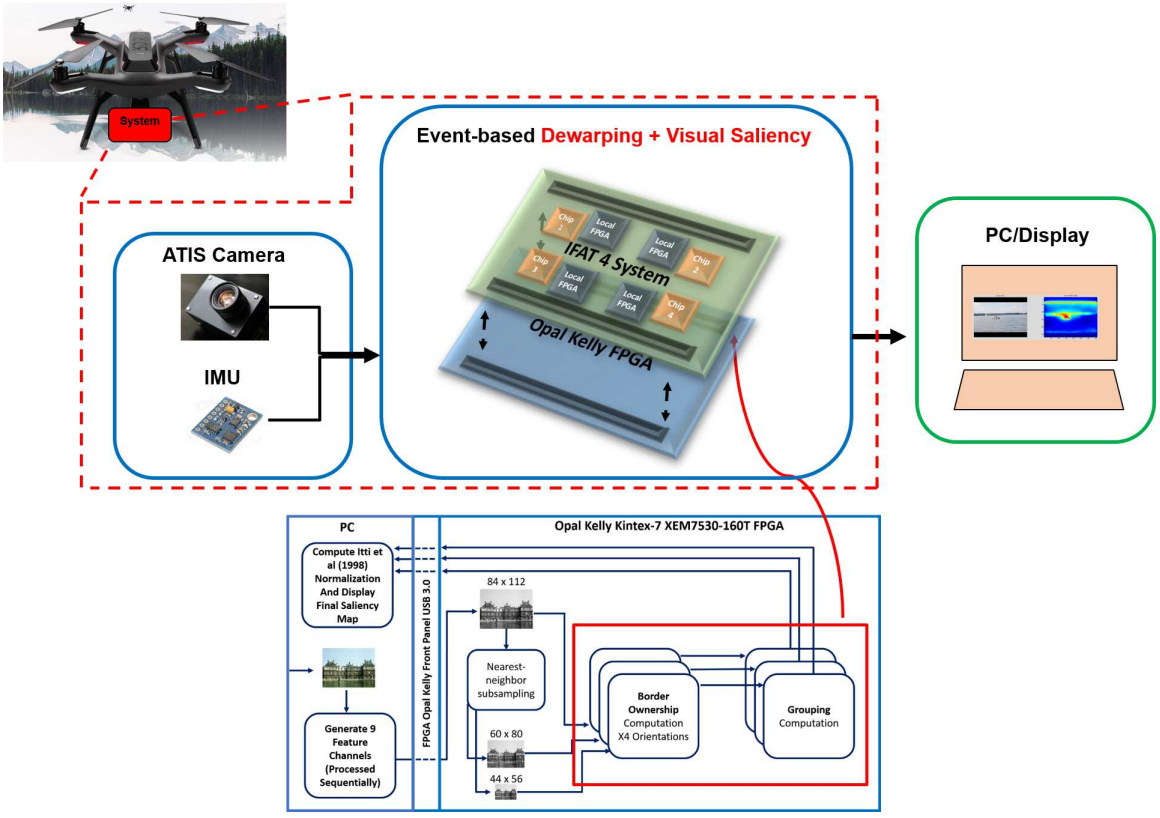


Figure 7.3: Multi-chip IFAT system (i.e. for simultaneous visual saliency and dewarping task via drone).

7.2 Summary - Fusing It All Together

In this thesis, novel neuromorphic systems for performing visual pre-processing tasks were introduced, further merging the gap between the human brain and engineered systems. We first discussed a novel dynamic proto-object based visual saliency model which predicts human eye fixations on video better than other state-of-the-art models. Building on this model, its hardware implementation on FPGA for real-time processing was introduced. To our knowledge, this is the first proto-object based visual saliency model implemented on hardware for real-time processing. We then discussed a generalized neuromorphic processor called the Integrate-and-Fire Array Transceiver. We demonstrated an FPGA implementation of the IFAT and its capability of performing visual tasks like those used within the visual saliency model. Such a hardware implementation of a neuromorphic system takes us one step closer to the low-power, small-size, light-weight specifications we seek. Along these lines, we discussed a novel array of Mihalas-Niebur neurons implemented in VLSI technology. The design approach was optimized for high neuron density and low power consumption. We finally demonstrated a similar neural array of I&F neurons implemented in a 55nm technology which took advantage of a novel automated AER architecture. Both of these VLSI-based neural arrays were capable of performing the necessary visual pre-processing tasks including dewarping and filtering.

Although we discussed each of these systems independently, they are all interrelated. These topics can together be summarized as follows:

> "The design of neuromorphic systems (engineered systems inspired by neuroscience) in hardware for performing visual pre-processing tasks under low-power, light-weight, small-size and real-time constraints. Such visual pre-processing tasks include visual saliency, dewarping, and filtering."

These tasks all act as precursors to higher level visual tasks such as object tracking, object recognition, or even deep learning. Therefore, the work presented in this thesis can be useful for applications for mobile robots such as drones or other self-driving vehicles performing various visual tasks.

Bibliography

[1] A. F. Russell, S. Mihalaş, R. von der Heydt, E. Niebur, and R. Etienne-Cummings, "A model of proto-object based saliency," *Vision research*, vol. 94, pp. 1–15, 2014.

[2] "FSU:human vision and color perception," https://micro.magnet.fsu.edu/optics/ lightandcolor/vision.html, 2016-12-06.

[3] "The Retina Reference:normal retinal anatomy," http://www.retinareference.com/anatomy, 2016-12-06.

[4] D. Purves, G. J. Augustine, D. Fitzpatrick, W. C. Hall, A.-S. LaMantia, and L. E. White, "Neuroscience," *Sunderland, MA: Sinauer Associates*, vol. 5, 2012.

[5] "NYU:perception lecture notes retinal ganglion cells," http://www.cns.nyu.edu/ david/courses/perception/lecturenotes/ganglion/ ganglion.html, 2016-12-06.

[6] "BrainMind:neuroscience," http://brainmind.com/BrainLecture9.html, 2016-12-06.

[7] "MDP: Fast image filtering using the caching extension," http://mdp-toolkit.sourceforge.net/examples/convolution/image_convolution.html, 2016-12-06.

[8] R. A. Rensink, "The dynamic representation of scenes," *Visual cognition*, vol. 7, no. 1-3, pp. 17–42, 2000.

[9] R. Kimchi, Y. Yeshurun, and A. Cohen-Savransky, "Automatic, stimulus-driven attentional capture by objecthood," *Psychonomic Bulletin & Review*, vol. 14, no. 1, pp. 166–172, 2007.

[10] y. . . Parkhurst, Derrick, "Selective attention in natural vision: using computational models to quantify stimulus-driven attentional allocation," Ph.D. dissertation, The Johns Hopkins University.

[11] A. Calimera, E. Macii, and M. Poncino, "The human brain project and neuromorphic computing," *Functional neurology*, vol. 28, no. 3, pp. 191–196, 2013.

[12] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J.-M. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. A. Merolla, and K. Boahen, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 699–716, 2014.

[13] M. Mahowald, *An analog VLSI system for stereoscopic vision.* Springer Science & Business Media, 1994, vol. 265.

[14] D. H. Goldberg, G. Cauwenberghs, and A. G. Andreou, "Probabilistic synaptic weighting in a reconfigurable network of vlsi integrate-and-fire neurons," *Neural Networks*, vol. 14, no. 6, pp. 781–793, 2001.

[15] P. Lichtsteiner and T. Delbruck, "A 64x64 aer logarithmic temporal derivative silicon retina," *Research in Microelectronics and Electronics*, vol. 2, pp. 202–205, 2005.

[16] C. Posch, D. Matolin, and R. Wohlgenannt, "A qvga 143 db dynamic range frame-free pwm image sensor with lossless pixel-level video compression and time-domain cds," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 259–275, 2011.

[17] R. Jain, R. Kasturi, and B. G. Schunck, *Machine vision.* McGraw-Hill New York, 1995, vol. 5.

[18] L. Itti, C. Koch, E. Niebur *et al.*, "A model of saliency-based visual attention for rapid scene analysis," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 20, no. 11, pp. 1254–1259, 1998.

[19] H. Zhou, H. S. Friedman, and R. Von Der Heydt, "Coding of border ownership

in monkey visual cortex," *The Journal of Neuroscience*, vol. 20, no. 17, pp. 6594–6611, 2000.

[20] E. Craft, H. Schütze, E. Niebur, and R. Von Der Heydt, "A neural model of figure–ground organization," *Journal of neurophysiology*, vol. 97, no. 6, pp. 4310–4326, 2007.

[21] L. Itti, "Quantifying the contribution of low-level saliency to human eye movements in dynamic scenes," *Visual Cognition*, vol. 12, no. 6, pp. 1093–1123, 2005.

[22] J. L. Molin and R. Etienne-Cummings, "Live demonstration: Real-time implementation of a proto-object-based dynamic visual saliency model," in *Biomedical Circuits and Systems Conference (BioCAS), 2015 IEEE*. IEEE, 2015, pp. 1–1.

[23] J. L. Molin, T. Figliolia, K. Sanni, I. Doxas, A. Andreou, and R. Etienne-Cummings, "Fpga emulation of a spike-based, stochastic system for real-time image dewarping," in *Circuits and Systems (MWSCAS), 2015 IEEE 58th International Midwest Symposium on*. IEEE, 2015, pp. 1–4.

[24] X. Clady, C. Clercq, S.-H. Ieng, F. Houseini, M. Randazzo, L. Natale, C. Bartolozzi, and R. Benosman, "Asynchronous visual event-based time-to-contact," *Neuromorphic Engineering Systems and Applications*, vol. 51, 2015.

[25] J. L. Molin, J. Rattray, and R. Etienne-Cummings, "Stochastic image processing and simultaneous dewarping for aerial vehicles," in *Circuits and Systems (ISCAS), 2016 IEEE International Symposium on.* IEEE, 2016, pp. 2086–2089.

[26] J. Schemmel, D. Briiderle, A. Griibl, M. Hock, K. Meier, and S. Millner, "A wafer-scale neuromorphic hardware system for large-scale neural modeling," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems.* IEEE, 2010, pp. 1947–1950.

[27] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. S. Modha, "A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm," in *2011 IEEE custom integrated circuits conference (CICC).* IEEE, 2011, pp. 1–4.

[28] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The spinnaker project," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 652–665, 2014.

[29] J. Molin, A. Eisape, C. S. Thakur, V. Varghese, C. Brandli, and R. Etienne-Cummings, "Low-power, low-mismatch, highly-dense, array of vlsi mihalas-niebur neurons," *In Review*, 2017.

[30] V. Varghese, J. L. Molin, C. Brandli, S. Chen, and R. E. Cummings, "Dynamically reconfigurable silicon array of generalized integrate-and-fire neurons," in

Biomedical Circuits and Systems Conference (BioCAS), 2015 IEEE. IEEE, 2015, pp. 1–4.

[31] K. L. Bigos, A. R. Hariri, and D. R. Weinberger, *Neuroimaging Genetics: Principles and Practices*. Oxford University Press, 2016.

[32] K. P. Cosgrove, C. M. Mazure, and J. K. Staley, "Evolving knowledge of sex differences in brain structure, function, and chemistry," *Biological psychiatry*, vol. 62, no. 8, pp. 847–855, 2007.

[33] G. E. Moore, "Cramming more components onto integrated circuits. electronics, 38 (8), april 1965," *Electronics*, vol. 38, no. 8, 1965.

[34] R. R. Murphy, *Disaster robotics*. MIT press, 2014.

[35] S. Weiss, D. Scaramuzza, and R. Siegwart, "Monocular-slam–based navigation for autonomous micro helicopters in gps-denied environments," *Journal of Field Robotics*, vol. 28, no. 6, pp. 854–874, 2011.

[36] J. Markoff, "Google cars drive themselves, in traffic," *The New York Times*, vol. 10, no. A1, p. 9, 2010.

[37] J. Wei, J. M. Snider, J. Kim, J. M. Dolan, R. Rajkumar, and B. Litkouhi, "Towards a viable autonomous driving research platform," in *Intelligent Vehicles Symposium (IV), 2013 IEEE*. IEEE, 2013, pp. 763–770.

[38] M. Faessler, F. Fontana, C. Forster, E. Mueggler, M. Pizzoli, and D. Scaramuzza, "Autonomous, vision-based flight and live dense 3d mapping with a quadrotor micro aerial vehicle," *Journal of Field Robotics*, vol. 1, 2015.

[39] C. J. Bohil, B. Alicea, and F. A. Biocca, "Virtual reality in neuroscience research and therapy," *Nature reviews neuroscience*, vol. 12, no. 12, pp. 752–762, 2011.

[40] M. Chessa, F. Solari, and S. P. Sabatini, *Virtual reality to simulate visual tasks for robotic systems.* Citeseer, 2010.

[41] S. W. Kuffler, "Discharge patterns and functional organization of mammalian retina," *Journal of neurophysiology*, vol. 16, no. 1, pp. 37–68, 1953.

[42] L. M. Vaina and S. Soloviev, "First-order and second-order motion: neurological evidence for neuroanatomically distinct systems," *Progress in Brain Research*, vol. 144, pp. 197–212, 2004.

[43] L. Itti, "Visual salience," *Scholarpedia*, vol. 2, no. 9, p. 3327, 2007.

[44] K. Koch, J. McLean, M. Berry, P. Sterling, V. Balasubramanian, and M. A. Freed, "Efficiency of information transmission by retinal ganglion cells," *Current Biology*, vol. 14, no. 17, pp. 1523–1530, 2004.

[45] S. P. Strong, R. Koberle, R. R. d. R. van Steveninck, and W. Bialek, "Entropy and information in neural spike trains," *Physical review letters*, vol. 80, no. 1, p. 197, 1998.

[46] J. K. Tsotsos, "Is complexity theory appropriate for analyzing biological systems?" *Behavioral and Brain Sciences*, vol. 14, no. 04, pp. 770–773, 1991.

[47] A. M. Treisman and G. Gelade, "A feature-integration theory of attention," *Cognitive psychology*, vol. 12, no. 1, pp. 97–136, 1980.

[48] M. A. Basso and R. H. Wurtz, "Modulation of neuronal activity in superior colliculus by changes in target probability," *The Journal of Neuroscience*, vol. 18, no. 18, pp. 7519–7534, 1998.

[49] J. W. Bisley and M. E. Goldberg, "Neuronal activity in the lateral intraparietal area and spatial attention," *Science*, vol. 299, no. 5603, pp. 81–86, 2003.

[50] K. G. Thompson and N. P. Bichot, "A visual salience map in the primate frontal eye field," *Progress in brain research*, vol. 147, pp. 249–262, 2005.

[51] A. R. Koene and L. Zhaoping, "Feature-specific interactions in salience from combined feature contrasts: Evidence for a bottom–up saliency map in v1," *Journal of Vision*, vol. 7, no. 7, pp. 6–6, 2007.

[52] C. Koch and S. Ullman, "Shifts in selective visual attention: towards the underlying neural circuitry," in *Matters of intelligence*. Springer, 1987, pp. 115–141.

[53] R. Milanese, S. Gil, and T. Pun, "Attentive mechanisms for dynamic and static scene analysis," *Optical engineering*, vol. 34, no. 8, pp. 2428–2434, 1995.

[54] J. M. Wolfe, "Guided search 2.0 a revised model of visual search," *Psychonomic bulletin & review*, vol. 1, no. 2, pp. 202–238, 1994.

[55] E. Niebur and C. Koch, "Control of selective visual attention: Modeling the" where" pathway," 1996.

[56] J. Allman, F. Miezin, and E. McGuinness, "Stimulus specific responses from beyond the classical receptive field: neurophysiological mechanisms for local-global comparisons in visual neurons," *Annual review of neuroscience*, vol. 8, no. 1, pp. 407–430, 1985.

[57] K. Koffka, *Principles of Gestalt psychology*. Routledge, 2013, vol. 44.

[58] U. Neisser, *Cognitive psychology: Classic edition*. Psychology Press, 2014.

[59] J. Duncan, "Selective attention and the organization of visual information." *Journal of Experimental Psychology: General*, vol. 113, no. 4, p. 501, 1984.

[60] D. Walther and C. Koch, "Modeling attention to salient proto-objects," *Neural networks*, vol. 19, no. 9, pp. 1395–1407, 2006.

[61] M. I. Posner, R. D. Rafal, L. S. Choate, and J. Vaughan, "Inhibition of return: Neural basis and function," *Cognitive neuropsychology*, vol. 2, no. 3, pp. 211–228, 1985.

[62] D. Mahapatra, S. Winkler, and S.-C. Yen, "Motion saliency outweighs other

low-level features while watching videos," International Society for Optics and Photonics, pp. 68 060P–68 060P, 2008.

[63] R. L. De Valois, N. P. Cottaris, L. E. Mahon, S. D. Elfar, and J. A. Wilson, "Spatial and temporal receptive fields of geniculate and cortical cells and directional selectivity," *Vision research*, vol. 40, no. 27, pp. 3685–3702, 2000.

[64] C. Mead, "Neuromorphic electronic systems," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1629–1636, 1990.

[65] M. Mahowald and R. Douglas, "A silicon neuron," *Nature*, vol. 354, no. 6354, pp. 515–518, 1991.

[66] M. A. C. Maher, S. P. Deweerth, M. A. Mahowald, and C. A. Mead, "Implementing neural architectures using analog vlsi circuits," *IEEE Transactions on circuits and systems*, vol. 36, no. 5, pp. 643–652, 1989.

[67] L. Watts, D. A. Kerns, R. F. Lyon, and C. A. Mead, "Improved implementation of the silicon cochlea," *IEEE Journal of Solid-state circuits*, vol. 27, no. 5, pp. 692–700, 1992.

[68] J. Lin and K. Boahen, "A delay-insensitive address-event link," in *Asynchronous Circuits and Systems, 2009. ASYNC'09. 15th IEEE Symposium on.* IEEE, 2009, pp. 55–62.

[69] E. Culurciello, R. Etienne-Cummings, and K. A. Boahen, "A biomorphic digital

image sensor," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 2, pp. 281–294, 2003.

[70] M. Koyanagi, Y. Nakagawa, K.-W. Lee, T. Nakamura, Y. Yamada, K. Inamura, K.-T. Park, and H. Kurino, "Neuromorphic vision chip fabricated using three-dimensional integration technology," in *Solid-State Circuits Conference, 2001. Digest of Technical Papers. ISSCC. 2001 IEEE International.* IEEE, 2001, pp. 270–271.

[71] A. Moini, *Vision chips.* Springer Science & Business Media, 2012, vol. 526.

[72] R. Etienne-Cummings, J. Van der Spiegel, P. Mueller, and M.-Z. Zhang, "A foveated silicon retina for two-dimensional tracking," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 47, no. 6, pp. 504–517, 2000.

[73] T. Delbruck, "Silicon retina with correlation-based, velocity-tuned pixels," *IEEE Transactions on Neural Networks*, vol. 4, no. 3, pp. 529–541, 1993.

[74] K. A. Boahen and A. G. Andreou, "A contrast sensitive silicon retina with reciprocal synapses," 1992.

[75] G. Orchard, J. G. Martin, R. J. Vogelstein, and R. Etienne-Cummings, "Fast neuromimetic object recognition using fpga outperforms gpu implementations,"

IEEE transactions on neural networks and learning systems, vol. 24, no. 8, pp. 1239–1252, 2013.

[76] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. IEEE, 2010, pp. 257–260.

[77] R. J. Vogelstein, U. Mallik, J. T. Vogelstein, and G. Cauwenberghs, "Dynamically reconfigurable silicon array of spiking neurons with conductance-based synapses," *IEEE transactions on neural networks*, vol. 18, no. 1, pp. 253–265, 2007.

[78] J. Park, S. Ha, T. Yu, E. Neftci, and G. Cauwenberghs, "A 65k-neuron 73-mevents/s 22-pj/event asynchronous micro-pipelined integrate-and-fire array transceiver," in *2014 IEEE Biomedical Circuits and Systems Conference (BioCAS) Proceedings*. IEEE, 2014, pp. 675–678.

[79] S. Moradi and G. Indiveri, "A vlsi network of spiking neurons with an asynchronous static random access memory," in *2011 IEEE Biomedical Circuits and Systems Conference (BioCAS)*. IEEE, 2011, pp. 277–280.

[80] J. Harel, C. Koch, and P. Perona, "Graph-based visual saliency," in *Advances in neural information processing systems*, 2006, pp. 545–552.

[81] B. Han and B. Zhou, "High speed visual saliency computation on gpu," in *2007 IEEE International Conference on Image Processing*, vol. 1. IEEE, 2007, pp. I–361.

[82] P. Longhurst, K. Debattista, and A. Chalmers, "A gpu based saliency map for high-fidelity selective rendering," in *Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*. ACM, 2006, pp. 21–29.

[83] J. L. Molin, A. F. Russell, S. Mihalas, E. Niebur, and R. Etienne-Cummings, "Proto-object based visual saliency model with a motion-sensitive channel," in *2013 IEEE Biomedical Circuits and Systems Conference (BioCAS)*. IEEE, 2013, pp. 25–28.

[84] J. L. Molin, R. Etienne-Cummings, and E. Niebur, "How is motion integrated into a proto-object based visual saliency model?" in *Information Sciences and Systems (CISS), 2015 49th Annual Conference on*. IEEE, 2015, pp. 1–6.

[85] C. P. Brandli, "Ifat4 - a dynamic threshold integrate and fire neuron array transciever," *JHU Masters Thesis*, 2010.

[86] L. Itti and C. Koch, "Computational modelling of visual attention," *Nature reviews neuroscience*, vol. 2, no. 3, pp. 194–203, 2001.

[87] L. Itti and P. F. Baldi, "Bayesian surprise attracts human attention," in *Advances in neural information processing systems*, 2005, pp. 547–554.

[88] N. D. Bruce and J. K. Tsotsos, "Saliency, attention, and visual search: An information theoretic approach," *Journal of vision*, vol. 9, no. 3, pp. 5–5, 2009.

[89] J. Harel and C. Koch, "Perona. graph based visual saliency," in *Neural Information Processing Systems (NIPS)*, 2007, pp. 1–8.

[90] Z. Lingyun, M. H. Tong, and G. W. Cottrell, "Information attracts attention: A probabilistic account of the cross-race advantage in visual search," in *in Proceedings of the 29th Annual Cognitive Science Conference*. Citeseer, 2007.

[91] Y. Sun, R. Fisher, F. Wang, and H. M. Gomes, "A computer vision model for visual-object-based attention and eye movements," *Computer Vision and Image Understanding*, vol. 112, no. 2, pp. 126–142, 2008.

[92] V. Yanulevskaya, J. Uijlings, J.-M. Geusebroek, N. Sebe, and A. Smeulders, "A proto-object-based computational model for visual saliency," *Journal of vision*, vol. 13, no. 13, pp. 27–27, 2013.

[93] K. R. Cave and N. P. Bichot, "Visuospatial attention: Beyond a spotlight model," *Psychonomic bulletin & review*, vol. 6, no. 2, pp. 204–223, 1999.

[94] F. T. Qiu, T. Sugihara, and R. von der Heydt, "Figure-ground mechanisms

provide structure for selective attention," *Nature neuroscience*, vol. 10, no. 11, pp. 1492–1499, 2007.

[95] Z. Li, "Primary cortical dynamics for visual grouping," 1997.

[96] N. R. Zhang and R. von der Heydt, "Analysis of the context integration mechanisms underlying figure–ground organization in the visual cortex," *The Journal of Neuroscience*, vol. 30, no. 19, pp. 6482–6496, 2010.

[97] R. Rosenholtz, "A simple saliency model predicts a number of motion popout phenomena," *Vision research*, vol. 39, no. 19, pp. 3157–3163, 1999.

[98] D. Gao, V. Mahadevan, and N. Vasconcelos, "On the plausibility of the discriminant center-surround hypothesis for visual saliency," *Journal of vision*, vol. 8, no. 7, pp. 13–13, 2008.

[99] H. J. Seo and P. Milanfar, "Static and space-time visual saliency detection by self-resemblance," *Journal of vision*, vol. 9, no. 12, pp. 15–15, 2009.

[100] L. Itti and P. Baldi, "Bayesian surprise attracts human attention," *Vision research*, vol. 49, no. 10, pp. 1295–1306, 2009.

[101] L. Zhang, M. H. Tong, T. K. Marks, H. Shan, and G. W. Cottrell, "Sun: A bayesian framework for saliency using natural statistics," *Journal of vision*, vol. 8, no. 7, pp. 32–32, 2008.

[102] S. Marat, T. H. Phuoc, L. Granjon, N. Guyader, D. Pellerin, and A. Guérin-Dugué, "Modelling spatio-temporal saliency to predict gaze direction for short videos," *International journal of computer vision*, vol. 82, no. 3, pp. 231–243, 2009.

[103] C. Guo and L. Zhang, "A novel multiresolution spatiotemporal saliency detection model and its applications in image and video compression," *IEEE Transactions on Image Processing*, vol. 19, no. 1, pp. 185–198, 2010.

[104] T. Liu, Z. Yuan, J. Sun, J. Wang, N. Zheng, X. Tang, and H.-Y. Shum, "Learning to detect a salient object," *IEEE Transactions on Pattern analysis and machine intelligence*, vol. 33, no. 2, pp. 353–367, 2011.

[105] P. Cavanagh, "Attention-based motion perception," *Science*, vol. 257, no. 5076, pp. 1563–1565, 1992.

[106] M. Hawken, R. Shapley, and D. Grosof, "Temporal-frequency selectivity in monkey visual cortex," *Vis Neurosci*, vol. 13, no. 3, pp. 477–492, 1996.

[107] J. Kulikowski, S. Marčelja, and P. Bishop, "Theory of spatial position and spatial frequency relations in the receptive fields of simple cells in the visual cortex," *Biological cybernetics*, vol. 43, no. 3, pp. 187–198, 1982.

[108] E. H. Adelson and J. R. Bergen, "Spatiotemporal energy models for the perception of motion," *JOSA A*, vol. 2, no. 2, pp. 284–299, 1985.

[109] T. E. Boult, "Remote reality demonstration," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 1998, p. 966.

[110] C.-S. Bouganis, P. Y. Cheung, and L. Zhaoping, "Fpga-accelerated pre-attentive segmentation in primary visual cortex," in *2006 International Conference on Field Programmable Logic and Applications*. IEEE, 2006, pp. 1–6.

[111] Z. Li, "Visual segmentation by contextual influences via intra-cortical interactions in the primary visual cortex," *Network: computation in neural systems*, vol. 10, no. 2, pp. 187–212, 1999.

[112] S. Kestur, M. S. Park, J. Sabarad, D. Dantara, V. Narayanan, Y. Chen, and D. Khosla, "Emulating mammalian vision on reconfigurable hardware," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 2012, pp. 141–148.

[113] P. Akselrod, F. Zhao, I. Derekli, C. Farabet, B. Martini, Y. LeCun, and E. Culurciello, "Hardware accelerated visual attention algorithm," in *Information Sciences and Systems (CISS), 2011 45th Annual Conference on*. IEEE, 2011, pp. 1–6.

[114] B. Kim, H. Okuno, T. Yagi, and M. Lee, "Implementation of visual attention system using artificial retina chip and bottom-up saliency map model," in *In-*

ternational Conference on Neural Information Processing. Springer, 2011, pp. 416–423.

[115] A. Moradhasel, B. N. Araabi, S. M. Fakhraie, and M. N. Ahmadabadi, "Fast saliency map extraction from video: A hardware approach," in *Machine Vision and Image Processing (MVIP), 2013 8th Iranian Conference on*. IEEE, 2013, pp. 64–69.

[116] F. Barranco, J. Diaz, B. Pino, and E. Ros, "Real-time visual saliency architecture for fpga with top-down attention modulation," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 3, pp. 1726–1735, 2014.

[117] S. Bae, Y. C. P. Cho, S. Park, K. M. Irick, Y. Jin, and V. Narayanan, "An fpga implementation of information theoretic visual-saliency system and its optimization," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE, 2011, pp. 41–48.

[118] N. Bruce and J. Tsotsos, "Attention based on information maximization," *Journal of Vision*, vol. 7, no. 9, pp. 950–950, 2007.

[119] J. Li, M. D. Levine, X. An, X. Xu, and H. He, "Visual saliency based on scale-space analysis in the frequency domain," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 4, pp. 996–1010, 2013.

[120] F. Ruffier and N. Franceschini, "Visually guided micro-aerial vehicle: auto-

matic take off, terrain following, landing and wind reaction," in *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, vol. 3. IEEE, 2004, pp. 2339–2346.

[121] A. Steiner, R. Moeckel, R. Thurer, D. Floreano, T. Delbruck, and S.-C. Liu, "1khz 2d silicon retina motion sensor platform," in *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on.* IEEE, 2014, pp. 41–44.

[122] K. Sanni, G. Garreau, J. L. Molin, and A. G. Andreou, "Fpga implementation of a deep belief network architecture for character recognition using stochastic computation," in *Information Sciences and Systems (CISS), 2015 49th Annual Conference on.* IEEE, 2015, pp. 1–5.

[123] M. Krips, T. Lammert, and A. Kummert, "Fpga implementation of a neural network for a real-time hand tracking system," in *Electronic Design, Test and Applications, 2002. Proceedings. The First IEEE International Workshop on.* IEEE, 2002, pp. 313–317.

[124] S. Himavathi, D. Anitha, and A. Muthuramalingam, "Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization," *IEEE Transactions on Neural Networks*, vol. 18, no. 3, pp. 880–888, 2007.

[125] J. G. Eldredge and B. L. Hutchings, "Rrann: a hardware implementation of the backpropagation algorithm using reconfigurable fpgas," in *Neural Networks,*

1994. IEEE World Congress on Computational Intelligence., 1994 IEEE International Conference on, vol. 4. IEEE, 1994, pp. 2097–2102.

[126] B. Glackin, T. M. McGinnity, L. P. Maguire, Q. Wu, and A. Belatreche, "A novel approach for the implementation of large scale spiking neural networks on fpga hardware," in *International Work-Conference on Artificial Neural Networks*. Springer, 2005, pp. 552–563.

[127] A. Cassidy, S. Denham, P. Kanold, and A. Andreou, "Fpga based silicon spiking neural array," in *Biomedical Circuits and Systems Conference, 2007. BIOCAS 2007. IEEE*. IEEE, 2007, pp. 75–78.

[128] S. L. Bade and B. L. Hutchings, "Fpga-based stochastic neural networks-implementation," in *FPGAs for Custom Computing Machines, 1994. Proceedings. IEEE Workshop on*. IEEE, 1994, pp. 189–198.

[129] B. R. Gaines *et al.*, "Stochastic computing systems," *Advances in information systems science*, vol. 2, no. 2, pp. 37–172, 1969.

[130] A. Alaghi, C. Li, and J. P. Hayes, "Stochastic circuits for real-time image-processing applications," in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*. IEEE, 2013, pp. 1–6.

[131] H. Kim, A. Handa, R. Benosman, S.-H. Ieng, and A. J. Davison, "Simultaneous

mosaicing and tracking with an event camera," *J. Solid State Circ*, vol. 43, pp. 566–576, 2008.

[132] M. Luo, R. Wang, S. Guo, J. Wang, J. Zou, and R. Huang, "Impacts of random telegraph noise (rtn) on digital circuits," *IEEE Transactions on Electron Devices*, vol. 62, no. 6, pp. 1725–1732, 2015.

[133] J. Park, T. Yu, S. Joshi, C. Maier, and G. Cauwenberghs, "Hierarchical address event routing for reconfigurable large-scale neuromorphic systems," *IEEE Transactions on Neural Networks and Learning Systems*, 2016.

[134] N. Qiao, H. Mostafa, F. Corradi, M. Osswald, F. Stefanini, D. Sumislawska, and G. Indiveri, "A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128k synapses," *Frontiers in neuroscience*, vol. 9, p. 141, 2015.

[135] S. Schultz and M. Jabri, "Analogue vlsi'integrate-and-fire'neuron with frequency adaptation," *Electronics Letters*, vol. 31, no. 16, pp. 1357–1358, 1995.

[136] A. van Schaik, "Building blocks for electronic spiking neural networks," *Neural networks*, vol. 14, no. 6, pp. 617–628, 2001.

[137] G. Indiveri, "A low-power adaptive integrate-and-fire neuron circuit," in *Circuits and Systems, 2003. ISCAS'03. Proceedings of the 2003 International Symposium on*, vol. 4. IEEE, 2003, pp. IV–IV.

[138] E. M. Izhikevich, "Which model to use for cortical spiking neurons?" *IEEE transactions on neural networks*, vol. 15, no. 5, pp. 1063–1070, 2004.

[139] S. Saighi, J. Tomas, Y. Bornat, and S. Renaud, "A conductance-based silicon neuron with dynamically tunable model parameters," in *Neural Engineering, 2005. Conference Proceedings. 2nd International IEEE EMBS Conference on.* IEEE, 2005, pp. 285–288.

[140] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[141] R. FitzHugh, "Impulses and physiological states in theoretical models of nerve membrane," *Biophysical journal*, vol. 1, no. 6, pp. 445–466, 1961.

[142] C. Morris and H. Lecar, "Voltage oscillations in the barnacle giant muscle fiber," *Biophysical journal*, vol. 35, no. 1, pp. 193–213, 1981.

[143] R. Rose and J. Hindmarsh, "The assembly of ionic currents in a thalamic neuron i. the three-dimensional model," *Proceedings of the Royal Society of London B: Biological Sciences*, vol. 237, no. 1288, pp. 267–288, 1989.

[144] H. R. Wilson, "Simplified dynamics of human and mammalian neocortical neurons," *Journal of theoretical biology*, vol. 200, no. 4, pp. 375–388, 1999.

[145] E. M. Izhikevich, "Resonate-and-fire neurons," *Neural networks*, vol. 14, no. 6, pp. 883–894, 2001.

[146] N. Fourcaud-Trocmé, D. Hansel, C. Van Vreeswijk, and N. Brunel, "How spike generation mechanisms determine the neuronal response to fluctuating inputs," *Journal of Neuroscience*, vol. 23, no. 37, pp. 11 628–11 640, 2003.

[147] R. Brette and W. Gerstner, "Adaptive exponential integrate-and-fire model as an effective description of neuronal activity," *Journal of neurophysiology*, vol. 94, no. 5, pp. 3637–3642, 2005.

[148] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on neural networks*, vol. 14, no. 6, pp. 1569–1572, 2003.

[149] J. H. Wijekoon and P. Dudek, "Compact silicon neuron circuit with spiking and bursting behaviour," *Neural Networks*, vol. 21, no. 2, pp. 524–534, 2008.

[150] Ş. Mihalaş and E. Niebur, "A generalized linear integrate-and-fire neural model produces diverse spiking behaviors," *Neural computation*, vol. 21, no. 3, pp. 704–718, 2009.

[151] F. Folowosele, T. J. Hamilton, and R. Etienne-Cummings, "Silicon modeling of the mihalaş–niebur neuron," *IEEE transactions on neural networks*, vol. 22, no. 12, pp. 1915–1927, 2011.

[152] F. Folowosele, A. Harrison, A. Cassidy, A. G. Andreou, R. Etienne-Cummings,

S. Mihalas, E. Niebur, and T. J. Hamilton, "A switched capacitor implementation of the generalized linear integrate-and-fire neuron," in *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on.* IEEE, 2009, pp. 2149–2152.

[153] A. Van Schaik, C. Jin, A. McEwan, T. J. Hamilton, S. Mihalas, and E. Niebur, "A log-domain implementation of the mihalas-niebur neuron model," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on.* IEEE, 2010, pp. 4249–4252.

[154] T. J. Hamilton and A. van Schaik, "Silicon implementation of the generalized integrate-and-fire neuron model," in *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2011 Seventh International Conference on.* IEEE, 2011, pp. 108–112.

[155] G. Indiveri, E. Chicca, and R. Douglas, "A vlsi array of low-power spiking neurons and bistable synapses with spike-timing dependent plasticity," *IEEE transactions on neural networks*, vol. 17, no. 1, pp. 211–221, 2006.

[156] M. A. Sivilotti, "Wiring considerations in analog vlsi systems, with application to field-programmable networks," Ph.D. dissertation, California Institute of Technology, 1990.

[157] K. A. Boahen, "Point-to-point connectivity between neuromorphic chips using

address events," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 47, no. 5, pp. 416–434, 2000.

[158] C. Sandor, A. Cunningham, A. Dey, and V.-V. Mattila, "An augmented reality x-ray system based on visual saliency," in *Mixed and Augmented Reality (ISMAR), 2010 9th IEEE International Symposium on.* IEEE, 2010, pp. 27–36.

[159] V. Sitzmann, A. Serrano, A. Pavel, M. Agrawala, D. Gutierrez, and G. Wetzstein, "Saliency in vr: How do people explore virtual environments?" *arXiv preprint arXiv:1612.04335*, 2016.

[160] S. Moradi and G. Indiveri, "An event-based neural network architecture with an asynchronous programmable synaptic memory," *IEEE transactions on biomedical circuits and systems*, vol. 8, no. 1, pp. 98–107, 2014.

Vita

Jamal Lottier Molin received his B.S. in Computer Engineering from the University of Maryland Baltimore County (UMBC) in May 2011. He attended UMBC as a Meyerhoff Scholar, a highly-selective scholarship program that seeks to increase the number of minorities pursuing a Ph.D. in STEM-related fields. While attending UMBC, he was also inducted into the Tau Beta Pi Honor Society (2010) and Omicron Delta Kappa National Leadership Honor Society (2011). He received his M.S.E in Electrical and Computer Engineering from Johns Hopkins University in December 2014. As a graduate student, he received the NeuroEngineering Training Initiative (NETI) Fellowship in 2013. He was awarded the DoD SMART (Department of Defense Science, Mathematics, and Research for Transformation) Fellowship in 2015. Through this fellowship he will be conducting research at the Naval Surface Warfare Center in Corona, CA. His research involves the design of neuromorphic systems for performing event-based visual processing un-

der low-power, small-size, light-weight constraints. It further involves understanding motion and how it can be applied to a proto-object-based visual saliency model, and further, implementing this dynamic visual saliency model in hardware (i.e. FPGA, VLSI) using these neuromorphic systems.